MATLAB®

The Language of Technical Computing

- Computation
- Visualization
- Programming

Creating Graphical User Interfaces

Version 7

How to Contact The MathWorks:

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

MATLAB Creating Graphical User Interfaces

© COPYRIGHT 2000 - 2006 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks of The MathWorks, Inc. Other product or brand names are trademarks or registered trademarks of their respective holders.

Patents

The MathWorks products are protected by one or more U.S. patents. Please see `www.mathworks.com/patents` for more information.

Revision History

November 2000	Online only	New for MATLAB 6.0 (Release12)
June 2001	Online only	Revised for MATLAB 6.1 (Release 12.1)
July 2002	Online only	Revised for MATLAB 6.5 (Release 13)
June 2004	Online only	Revised for MATLAB 7.0 (Release 14)
October 2004	Online only	Revised for MATLAB 7.0.1 (Release 14SP1)
March 2005	Online only	Revised for MATLAB 7.0.4 (Release 14SP2)
September 2005	Online only	Revised for MATLAB 7.1 (Release 14SP3)
March 2006	Online only	Revised for MATLAB 7.2 (Release 2006a)

Contents

Getting Started

1 About GUIs in MATLAB

What Is a GUI? **1-2**

How Does a GUI Work? **1-3**

Where Do I Start? **1-4**

2 Creating a GUI with GUIDE

GUIDE: A Brief Introduction **2-2**
- Laying Out a GUI **2-2**
- Programming the GUI **2-2**

The Example: Simple GUI **2-3**

Laying Out a Simple GUI **2-4**
- Opening a New GUI in the Layout Editor **2-4**
- Setting the GUI Figure Size **2-6**
- Adding the Components **2-7**
- Aligning the Components **2-9**
- Adding Text to the Components **2-11**

Saving the GUI Layout **2-17**

Programming the GUI **2-19**

Generating Data to Plot 2-19
Programming the Pop-Up Menu 2-21
Programming the Push Buttons 2-23

Running the GUI 2-26

3 Creating a GUI Programmatically

The Example: Simple GUI 3-2

Function Summary 3-4
Functions Used to Create the Simple GUI 3-4
Other MATLAB Functions Used in the GUI 3-4

Creating a GUI M-File 3-6

Laying Out a Simple GUI 3-7
Creating the Figure 3-7
Adding the Components 3-7

Initializing the GUI 3-11

Programming the GUI 3-14
Programming the Pop-Up Menu 3-14
Programming the Push Buttons 3-15
Associating Callbacks with Their Components 3-15

Running the Final GUI 3-17
The Final M-File 3-17
Running the GUI 3-19

Creating GUIs with GUIDE

4 What Is GUIDE?

What Is GUIDE? **4-2**
Laying Out a GUI **4-2**
Programming the GUI **4-2**

GUIDE Tools Summary **4-3**

5 Preferences and Options

GUIDE Preferences **5-2**
Confirmation Preferences **5-2**
Backward Compatibility Preference **5-4**
All Other Preferences **5-5**

GUI Options **5-9**
Resize Behavior **5-9**
Command-Line Accessibility **5-10**
Generate FIG-File and M-File **5-11**
Generate FIG-File Only **5-13**

6 Laying Out a GUI

Designing a GUI **6-2**

Starting GUIDE **6-4**

Selecting a GUI Template **6-5**
Accessing the Templates **6-5**
Template Descriptions **6-6**

Setting the GUI Size **6-12**

Adding Components to the GUI **6-14**
Available Components **6-14**
Adding Components to the Layout Area **6-18**
Assigning an Identifier to Each Component **6-24**
Adding Text to the Components **6-25**
Working with Components in the Layout Area **6-31**
Moving Components **6-34**
Resizing Components **6-37**

Aligning Components **6-39**
Alignment Tool **6-39**
Property Inspector **6-41**
Grid and Rulers **6-42**
Guide Lines **6-43**

Setting Tab Order **6-45**

Creating Menus **6-48**
Menus for the Menu Bar **6-49**
Context Menus **6-56**

Creating Toolbars **6-61**

Viewing the Object Hierarchy **6-62**

Designing for Cross-Platform Compatibility **6-63**
Default System Font **6-63**
Standard Background Color **6-64**
Cross-Platform Compatible Units **6-66**

7 Saving and Running the GUI

Naming a GUI and Its Files 7-2
The GUI Files 7-2
File and GUI Names 7-3
How to Assign Names 7-3
Renaming GUIs and GUI Files 7-4

Saving a GUI 7-5
Ways to Save a GUI 7-5
Saving a New GUI 7-7
Saving an Existing GUI 7-9

Running a GUI 7-10
From the GUIDE Layout Editor 7-10
From the Command Line 7-10
From an M-file Script 7-11

Saving and Displaying Only the GUI Layout 7-12
Saving a GUI Layout Only 7-12
Displaying a GUI Layout Only 7-12

8 Programming the GUI

Callbacks: An Overview 8-2
What Is a Callback? 8-2
Kinds of Callbacks 8-2

GUI Files: An Overview 8-5
M-Files and FIG-Files 8-5
GUI M-File Structure 8-6
Adding Callback Templates to an Existing GUI M-File 8-7

Associating Callbacks with Components 8-8
Setting Callback Properties Automatically 8-8

Callback Syntax and Arguments **8-11**
Naming of Callback Functions **8-11**
Changing Callback Names Assigned by GUIDE **8-12**
Input Arguments **8-13**
The Handles Structure **8-13**

Initialization Callbacks **8-15**
Opening Function **8-15**
Output Function **8-17**

Examples: Programming GUI Components **8-19**
Push Button **8-19**
Toggle Button **8-20**
Radio Button **8-21**
Check Box **8-21**
Edit Text **8-22**
Slider **8-23**
List Box **8-23**
Pop-Up Menu **8-24**
Panel **8-25**
Button Group **8-26**
Axes **8-27**
ActiveX Control **8-31**
Menu Item **8-37**

9 Managing Application-Defined Data

Mechanisms for Managing Data **9-2**
GUI Data **9-2**
Application Data **9-4**
UserData Property **9-6**

Sharing Data Among a GUI's Callbacks **9-7**
GUI Data **9-7**
Application Data **9-9**
UserData Property **9-11**

10 Examples of GUIDE GUIs

GUI with Multiple Axes **10-2**
Techniques Used in the Example **10-2**
View Completed Layout and Its GUI M-File **10-3**
Design of the GUI **10-3**
Plot Push Button Callback **10-6**

List Box Directory Reader **10-9**
View Layout and GUI M-File **10-9**
Implementing the GUI **10-10**
Specifying the Directory to List **10-10**
Loading the List Box **10-12**

Accessing Workspace Variables from a List Box **10-16**
Techniques Used in This Example **10-16**
View Completed Layout and Its GUI M-File **10-17**
Reading Workspace Variables **10-17**
Reading the Selections from the List Box **10-17**

A GUI to Set Simulink Model Parameters **10-20**
Techniques Used in This Example **10-20**
View Completed Layout and Its GUI M-File **10-20**
How to Use the GUI (Text of GUI Help) **10-21**
Running the GUI **10-22**
Programming the Slider and Edit Text Components **10-23**
Running the Simulation from the GUI **10-25**
Removing Results from the List Box **10-27**
Plotting the Results Data **10-28**
The GUI Help Button **10-29**
Closing the GUI **10-30**
The List Box Callback and Create Function **10-30**

An Address Book Reader **10-32**
Techniques Used in This Example **10-32**
Managing Shared Data **10-32**
View Completed Layout and Its GUI M-File **10-33**
Running the GUI **10-33**
Loading an Address Book Into the Reader **10-35**

The Contact Name Callback **10-38**
The Contact Phone Number Callback **10-39**
Paging Through the Address Book — Prev/Next **10-40**
Saving Changes to the Address Book from the Menu **10-42**
The Create New Menu **10-44**
The Address Book Resize Function **10-44**

Using a Modal Dialog to Confirm an Operation **10-47**
View Completed Layouts and Their GUI M-Files **10-47**
Setting Up the Close Confirmation Dialog **10-48**
Setting Up the GUI with the Close Button **10-49**
Running the GUI with the Close Button **10-50**
How the GUI and Dialog Work **10-51**

Creating GUIs Programmatically

11 Functions for Creating GUIs

Introduction 11-2

Function Summary 11-3
Predefined Dialog Boxes 11-3
Creating Components 11-3
ActiveX Controls 11-4
Using Application-Defined Data 11-4
Interactive User Input 11-4
Working with Preferences 11-4
Utilities 11-5

12 Examples of GUIs Created Programmatically

Introduction 8

GUI with Axes, Menu, and Toolbar 9
The Example 9
Techniques Used in the Example 11
View the Completed GUI M-Files 11
Creating the Data 11
Creating the GUI and Its Components 12
Initializing the GUI 17
Defining the Callbacks 18
Helper Function: Plotting the Plot Types 21

Color Palette 22
The Example 22

Techniques Used in the Example 25
View the Completed GUI M-File 25
Subfunction Summary 26
M-File Structure 27
GUI Programming Techniques 28

Icon Editor 33
The Example 33
Techniques Used in the Example 35
View the Completed GUI M-Files 35
Subfunction Summary 36
M-File Structure 37
GUI Programming Techniques 38

Index

Getting Started

These sections show you how to create a simple graphical user interface (GUI) using GUIDE, the MATLAB® graphical user interface development environment, and how to create the same GUI programmatically.

About GUIs in MATLAB (p. 1-1)	Explains what a GUI is, how a GUI works, and how to get started creating a GUI.
Creating a GUI with GUIDE (p. 2-1)	Steps you through the process of creating a simple GUI using GUIDE.
Creating a GUI Programmatically (p. 3-1)	Steps you through the process of creating a simple GUI programmatically.

Note Following the release of MATLAB version 7.2, Creating Graphical User Interfaces will be further updated and expanded. The HTML and PDF versions of this document will be updated on The MathWorks Web site some time after the release. Check this page and the title page of the PDF document to determine if they have been updated.

1

About GUIs in MATLAB

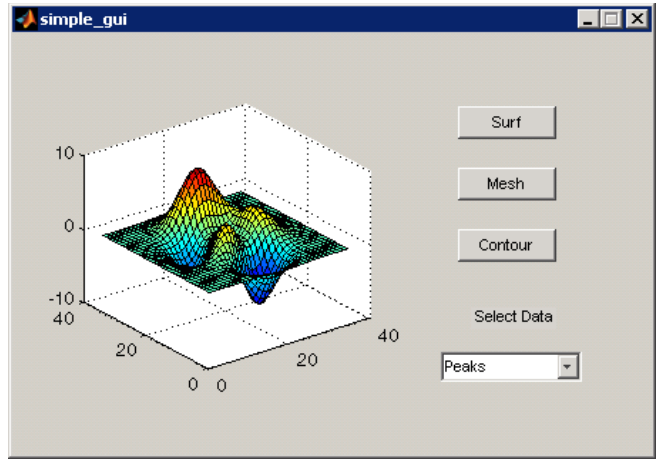
What Is a GUI? (p. 1-2)	Explains a graphical user interface (GUI) from a GUI user's perspective.
How Does a GUI Work? (p. 1-3)	Explains how a GUI operates from a software point of view.
Where Do I Start? (p. 1-4)	Describes different techniques for creating GUIs in MATLAB.

What Is a GUI?

A graphical user interface (GUI) is a graphical display that contains devices, or components, that enable a user to perform interactive tasks. To perform these tasks, the user of the GUI does not have to create a script or type commands at the command line. Often the user does not have to know the details of the task at hand.

The GUI components can be menus, toolbars, push buttons, radio buttons, list boxes, and sliders - just to name a few. In MATLAB, a GUI can also display data in tabular form or as plots, and can group related components.

The following figure illustrates a simple GUI.



The GUI contains

- An axes
- A pop-up menu listing three different data sets that correspond to MATLAB functions: peaks, membrane, and sinc
- A static text to label the pop-up menu
- Three buttons that provide different kinds of plots: surface, mesh, and contour

When you click a push button, the axes displays the selected data set using the specified plot.

How Does a GUI Work?

Each component, and the GUI itself, is associated with one or more user-written routines known as callbacks. The execution of each callback is triggered by a particular user action such as a button push, mouse click, selection of a menu item, or the cursor passing over a component. You, as the creator of the GUI, provide these callbacks.

In the GUI described in “What Is a GUI?” on page 1-2, the user selects a data set from the pop-up menu, then clicks one of the plot type buttons. Clicking the button triggers the execution of a callback that plots the selected data in the axes.

This kind of programming is often referred to as event-driven programming. The event in the example is a button click. In event-driven programming, callback execution is asynchronous, controlled by events external to the software. In the case of MATLAB GUIs, these events usually take the form of user interactions with the GUI.

The writer of a callback has no control over the sequence of events that leads to its execution or, when the callback does execute, what other callbacks might be running simultaneously.

Where Do I Start?

First you have to design your GUI. You have to decide what you want it to do, how you want the user to interact with it, and what components you need. “Designing a GUI” on page 6-2 list references that may be of help.

Next, you must decide what technique you want to use to create your GUI. MATLAB enables you to create GUIs programmatically or with GUIDE, an interactive GUI builder. It also provides functions that simplify the creation of standard dialog boxes. The technique you choose depends on your experience, your preferences, and the kind of GUI you want to create. This table outlines some possibilities.

GUI	Technique
Dialog box	MATLAB provides a selection of standard dialog boxes that you can create with a single function call. For links to these functions, see Predefined Dialog Boxes in the MATLAB documentation.
GUI containing just a few components	It is often simpler to create GUIs that contain only a few components programmatically. Each component can be fully defined with a single function call.
Moderately complex GUIs	GUIDE simplifies the creation of such GUIs.
Complex GUIs with many components, and GUIs that require interaction with other GUIs	Creating such GUIs programmatically lets you control exact placement of the components and provides reproducibility.

Once you have decided which technique you want to use, you can continue to learn about creating GUIs in MATLAB by following through the examples in the following topics:

- Chapter 2, “Creating a GUI with GUIDE”
- Chapter 3, “Creating a GUI Programmatically”

2

Creating a GUI with GUIDE

GUIDE: A Brief Introduction (p. 2-2)	Introduces GUIDE, the graphical user interface development environment.
The Example: Simple GUI (p. 2-3)	Describes the example to be constructed.
Laying Out a Simple GUI (p. 2-4)	Lays out the GUI's components, including moving, aligning, and labeling components.
Saving the GUI Layout (p. 2-17)	Saves the GUI and gives it a name.
Programming the GUI (p. 2-19)	Generates the data to plot and adds code for each component to the GUI M-file to make the GUI work.
Running the GUI (p. 2-26)	Runs the GUI and demonstrates how the components work together.

GUIDE: A Brief Introduction

GUIDE, the MATLAB Graphical User Interface development environment, provides a set of tools for creating graphical user interfaces (GUIs). These tools simplify the process of laying out and programming GUIs.

Laying Out a GUI

The GUIDE Layout Editor enables you to populate a GUI by clicking and dragging GUI components — such as buttons, text fields, sliders, axes, and so on — into the layout area. It also enables you to create menus and context menus for the GUI.

Other tools, which are accessible from the Layout Editor, enable you to size the GUI, modify component look and feel, align components, set tab order, view a hierarchical list of the component objects, and set GUI options.

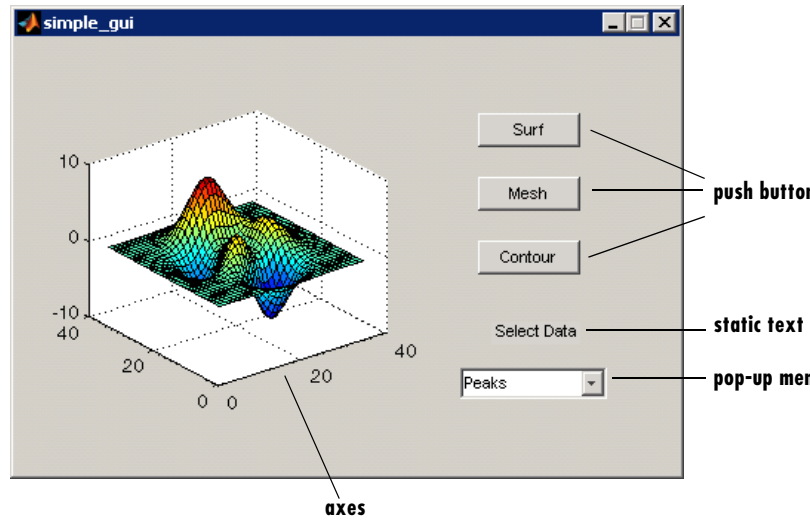
The following topic, “Laying Out a Simple GUI” on page 2-4,uses some of these tools to show you the basics of laying out a GUI. “GUIDE Tools Summary” on page 4-3 describes the tools.

Programming the GUI

When you save your GUI layout, GUIDE automatically generates an M-file that you can use to control how the GUI works. This M-file provides code to initialize the GUI and contains a framework for the GUI callbacks — the routines that execute in response to user-generated events such as a mouse click. Using the M-file editor, you can add code to the callbacks to perform the functions you want them to. “Programming the GUI” on page 2-19 shows you what code to add to the example M-file to make the GUI work.

The Example: Simple GUI

This chapter shows you how to use GUIDE to create the example graphical user interface (GUI) which is shown in the following figure.



The GUI contains:

- An axes
- A pop-up menu listing three different data sets that correspond to MATLAB functions: `peaks`, `membrane`, and `sinc`
- A static text to label the pop-up menu
- Three push buttons that provide different kinds of plots: surface, mesh, and contour

To use the GUI, the user selects a data set from the pop-up menu, then clicks one of the plot-type buttons. Clicking the button triggers the execution of a callback that plots the selected data in the axes.

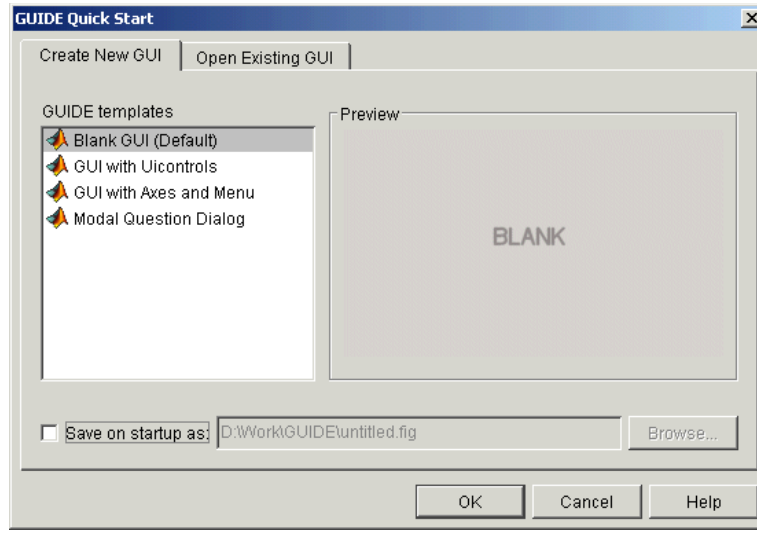
Subsequent topics GUIDE you through the steps to create this GUI. We recommend that you create the GUI for yourself, as this is the best way to learn how to use GUIDE.

Laying Out a Simple GUI

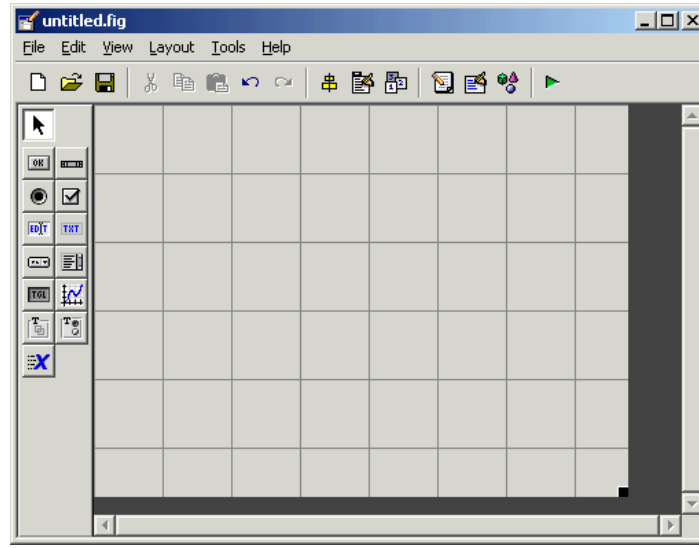
- "Opening a New GUI in the Layout Editor" on page 2-4
- "Setting the GUI Figure Size" on page 2-6
- "Adding the Components" on page 2-7
- "Aligning the Components" on page 2-9
- "Adding Text to the Components" on page 2-11

Opening a New GUI in the Layout Editor

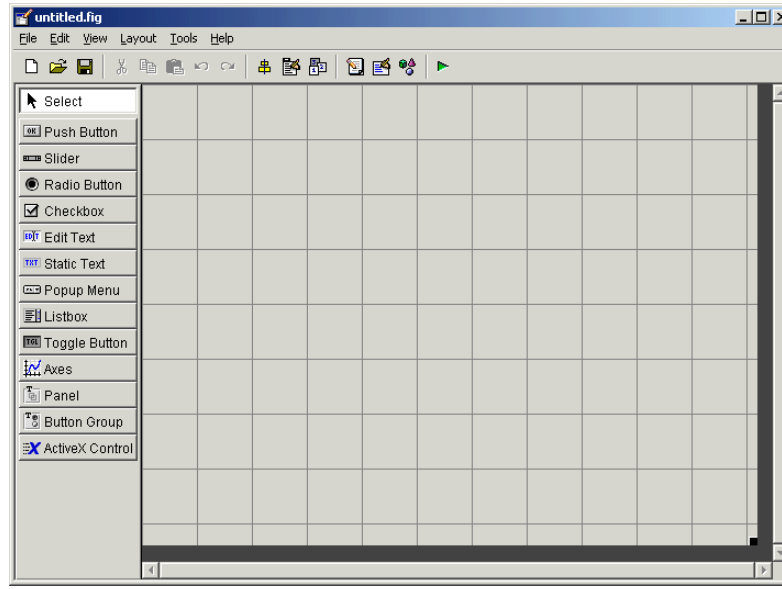
1 Start GUIDE by typing `guide` at the MATLAB prompt. This displays the **GUIDE Quick Start** dialog shown in the following figure.



2 In the Quick Start dialog, select the **Blank GUI (default)** template. Click **OK** to display the blank GUI in the Layout Editor, as shown in the following figure.

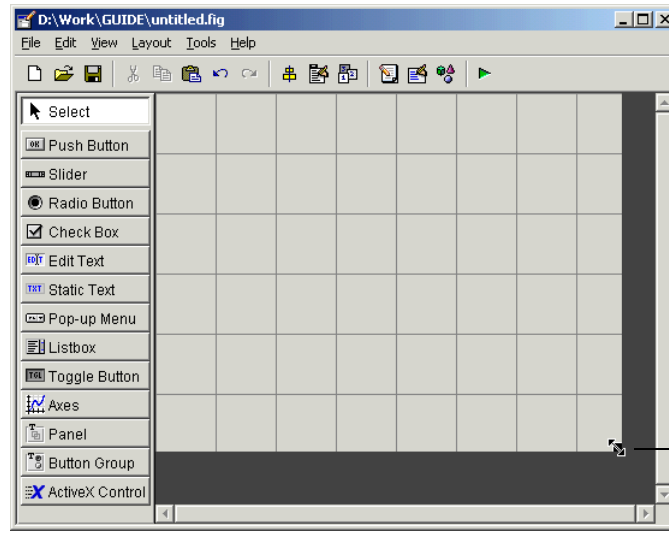


3 Display the names of the GUI components in the component palette. Select **Preferences** from the MATLAB **File** menu. Then select **GUIDE -> Show names in component palette**, and click **OK**. The Layout Editor then appears as shown in the following figure.



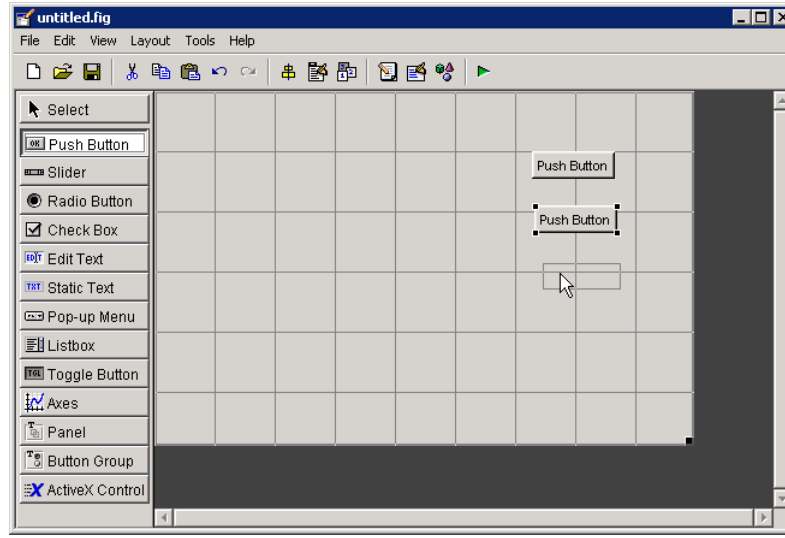
Setting the GUI Figure Size

Set the size of the GUI by resizing the grid area in the Layout Editor. Click on the lower-right corner and drag it until the GUI is approximately 3 inches high and 4 inches wide. If necessary, make the window larger.



Adding the Components

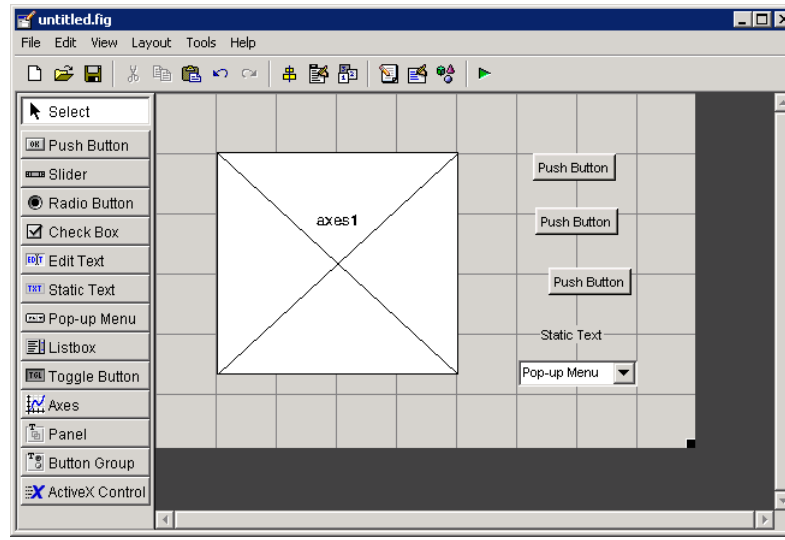
1 Add the three push buttons to the GUI. For each push button, select the push button from the component palette at the left of the Layout Editor and drag it into the layout area. Position them approximately as shown in the following figure.



2 Add the remaining components to the GUI.

- A static text
- A pop-up menu
- An axes

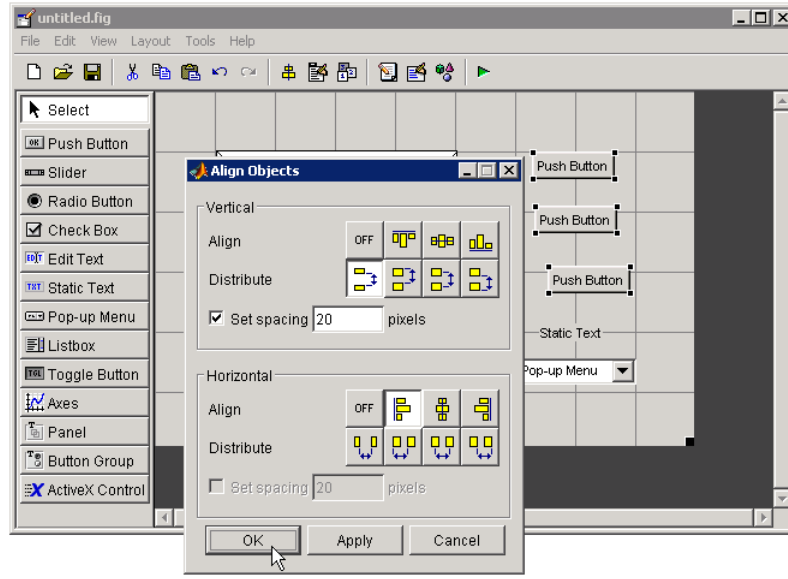
Arrange the components as shown in the following figure. Resize the axes component to approximately 2-by-2 inches.



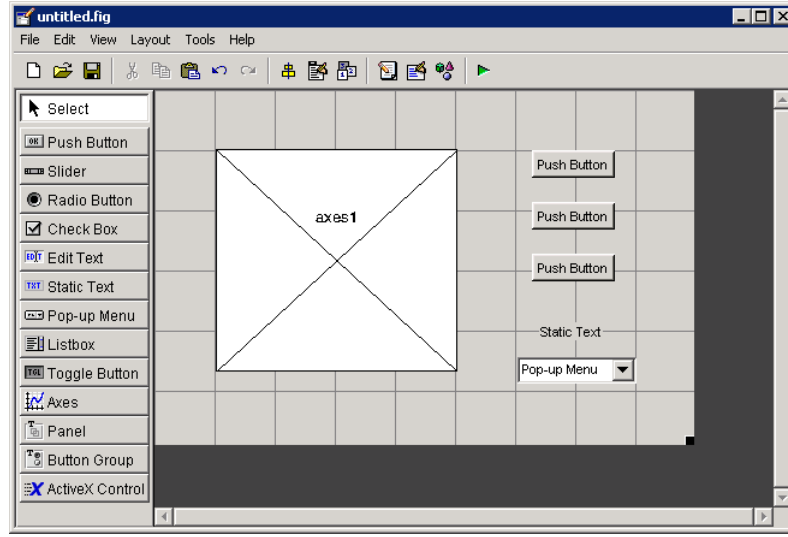
Aligning the Components

You can use the Alignment Tool to align components with respect to one another, if they have the same parent. To align the three push buttons:

1. Select all three push buttons by pressing **Ctrl** and clicking them.
2. Select **Align Objects** from the **Tools** menu to display the Alignment Tool.
3. Make these settings in the Alignment Tool, as shown in the following figure:
   - 20 pixels spacing between push buttons in the vertical direction.
   - Left-aligned in the horizontal direction.



4 Click **OK**. Your GUI now looks like this is the Layout Editor.

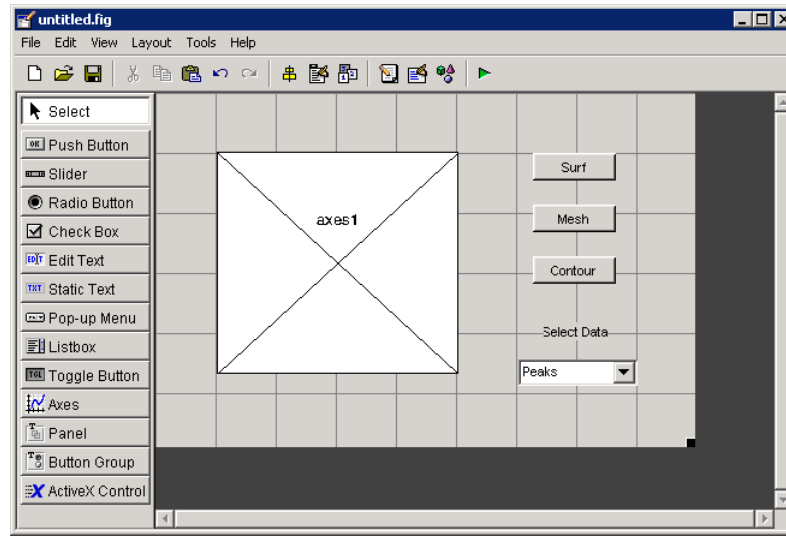


Adding Text to the Components

Although the push buttons, pop-up menu, and static text show some text in the Layout Editor, the text is not appropriate to the GUI being created. This topic shows you how to modify the default text.

- "Labeling the Push Buttons" on page 2-11
- "Entering Pop-Up Menu Items" on page 2-13
- "Modifying the Static Text" on page 2-14

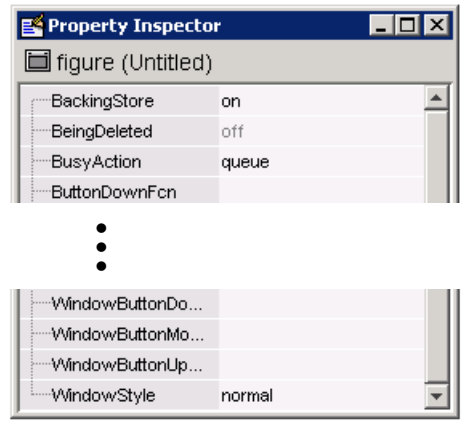
After you have added the appropriate text, the GUI will look like this in the Layout Editor.



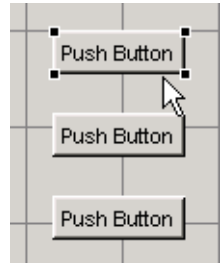
Labeling the Push Buttons

Each of the three push buttons lets the user choose a plot type: surf, mesh, and contour. This topic shows you how to label the buttons with those choices.

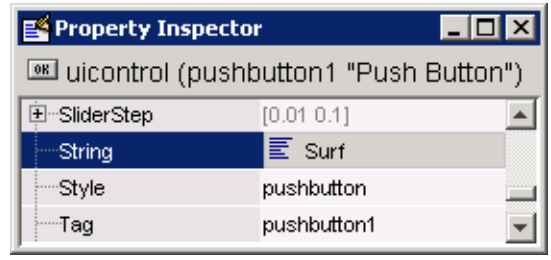
1 Select **Property Inspector** from the **View** menu.



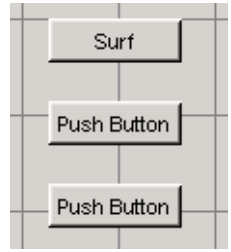
2 In the layout area select the top push button by clicking it.



3 In the Property Inspector, select the `String` property and then replace the existing value with the word `Surf`.



4 The push button label changes to **Surf** when you click outside the `String` field.



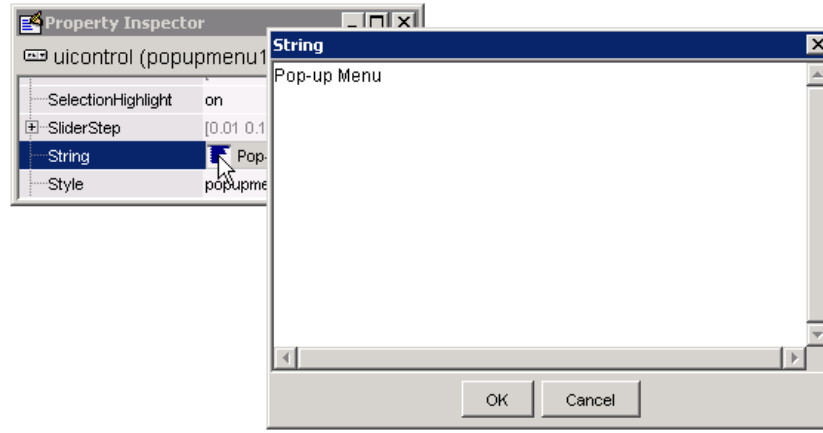
5 Select each of the remaining push buttons in turn and repeat steps 3 and 4. Label the middle push button **Mesh**, and the bottom button **Contour**.

Entering Pop-Up Menu Items

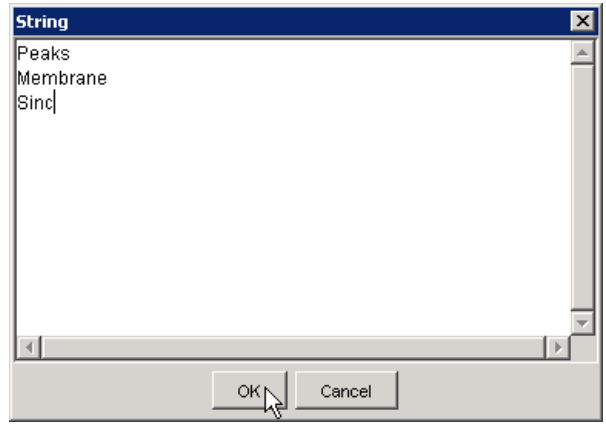
The pop-up menu provides a choice of three data sets: peaks, membrane, and sinc. These data sets correspond to MATLAB functions of the same name. This topic shows you how to list those data sets as choices in the pop-menu.

1 In the layout area, select the pop-up menu by clicking it.

2 In the Property Inspector, click the button next to `String`. The **String** dialog box displays.



3 Replace the existing text with the names of the three data sets: Peaks, Membrane, and Sinc. Press **Enter** to move to the next line. When you are done, click **OK**.



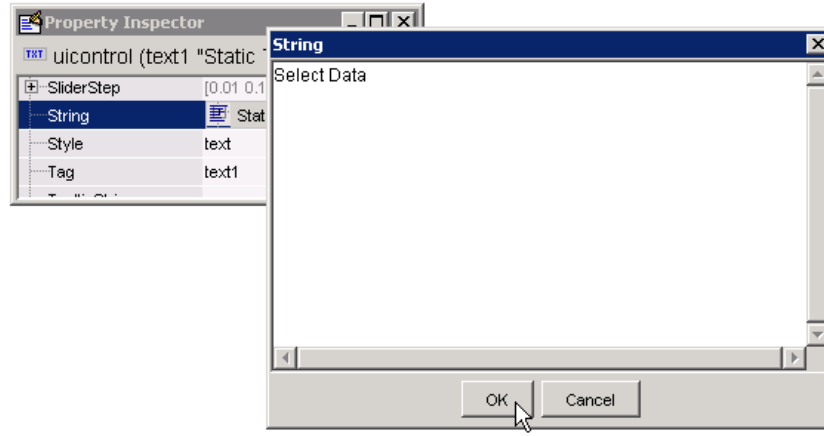
4 The first item in your list, `Peaks`, appears in the pop-up menu in the layout area.

Modifying the Static Text

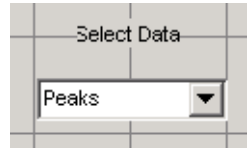
In this GUI, the static text serves as a label for the pop-up menu. The user cannot change this text. This topic shows you how to change the static text to read `Select Data`.

1 In the layout area, select the static text by clicking it.

2 In the Property Inspector, click the button next to `String`. In the **String** dialog box that displays, replace the existing text with the phrase `Select Data`, then click **OK**.

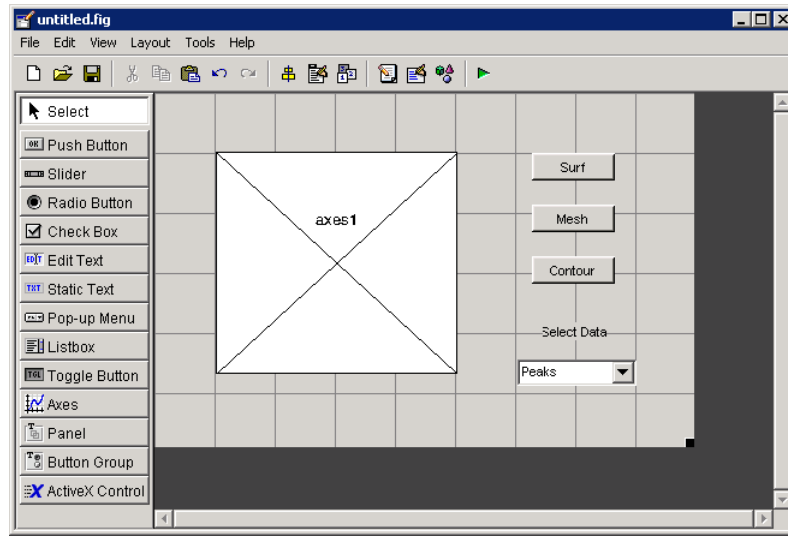


3 The phrase `Select Data` appears in the static text box in the layout area.



The Completed Layout

In the Layout Editor, your GUI now looks like this and the next step is to save the layout. The next topic, “Saving the GUI Layout” on page 2-17, tells you how to do this.

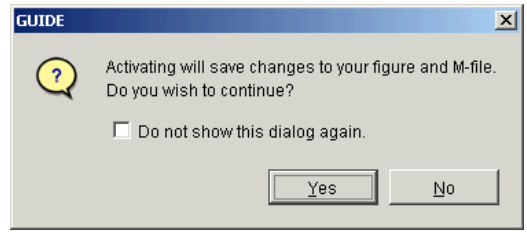


Saving the GUI Layout

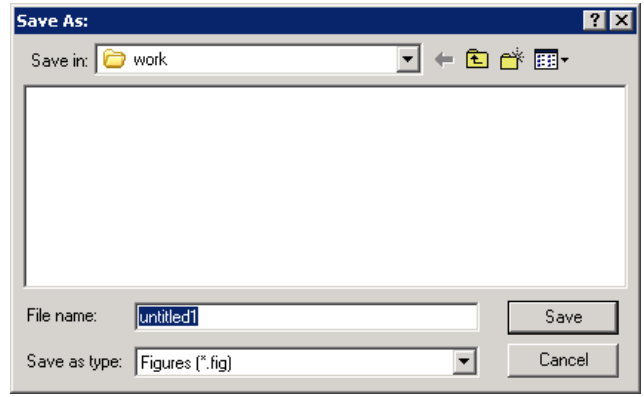
When you save a GUI, GUIDE creates two files, a FIG-file and an M-file. The FIG-file, with extension `.fig`, is a binary file that contains a description of the layout. The M-file, with extension `.m`, contains the code that controls the GUI.

1. Save and activate your GUI by selecting **Run** from the **Tools** menu.

2. GUIDE displays the following dialog box. Click **Yes** to continue.



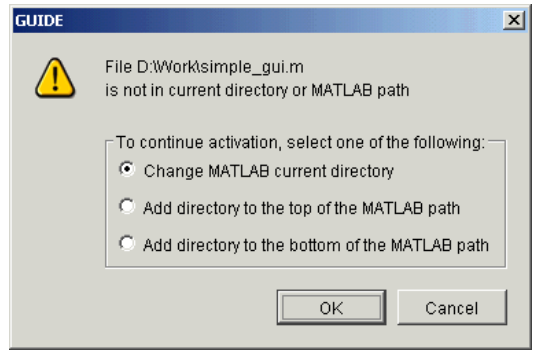
3. GUIDE opens a **Save As** dialog box in your current directory and prompts you for a FIG-file name.



4. Browse to any directory for which you have write privileges, and then enter the filename `simple_gui` for the FIG-file. GUIDE saves both the FIG-file and the M-file using this name.

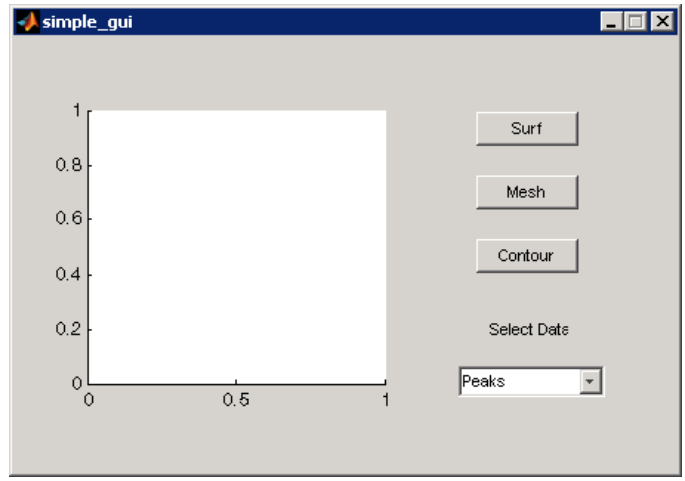
5. If the directory in which you save the GUI is not on the MATLAB path, GUIDE opens a dialog box, giving you the option of changing the current

working directory to the directory containing the GUI files, or adding that directory to the top or bottom of the MATLAB path.


GUIDE
File D:\Work\simple_gui.m
is not in current directory or MATLAB path
To continue activation, select one of the following:
Change MATLAB current directory
Add directory to the top of the MATLAB path
Add directory to the bottom of the MATLAB path
OK
Cancel

6 GUIDE saves the files `simple_gui.fig` and `simple_gui.m` and activates the GUI. It also opens the GUI M-file in your default editor.

The GUI is active. You can select a data set in the pop-up menu and click the push buttons. But nothing happens. This is because there is no code in the M-file to service the pop-up menu and the buttons. The next step is to program the GUI. The next topic, “Programming the GUI” on page 2-19, shows you how to do this.



Programming the GUI

When you saved your GUI in the previous topic, "Saving the GUI Layout" on page 2-17, GUIDE created two files: a FIG-file`simple_gui.fig` that contains the GUI layout, and an M-file `simple_gui.m` that contains the code that controls the GUI. But the GUI didn't do anything because there was no code in the M-file to make it work. This topic shows you how to add code to the M-file to make it work. There are three steps:

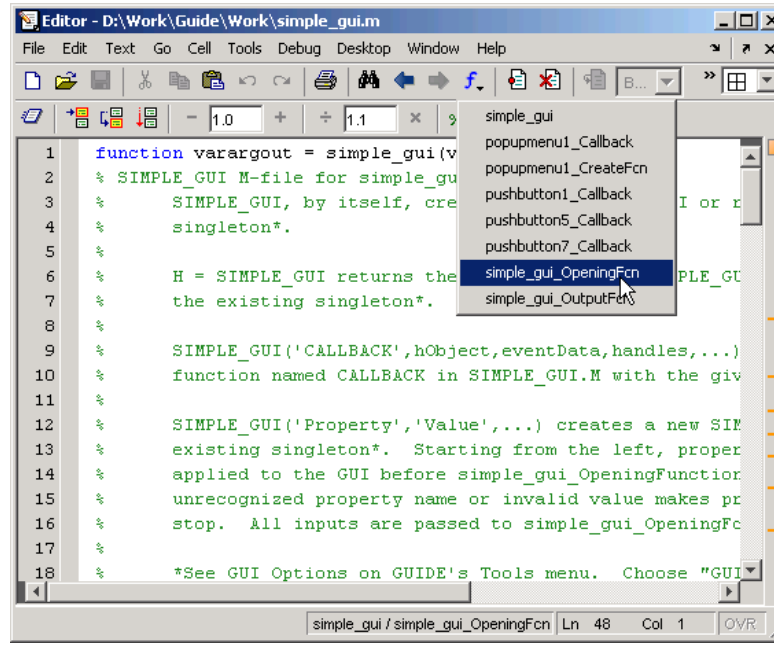
- "Generating Data to Plot" on page 2-19
- "Programming the Pop-Up Menu" on page 2-21
- "Programming the Push Buttons" on page 2-23

Generating Data to Plot

This topic shows you how to generate the data to be plotted when the user clicks a button. This data is generated in the *opening function*. The opening function is the first callback in every GUIDE-generated GUI M-file. You can use it to perform tasks that need to be done before the user has access to the GUI.

In this example, you add code that creates three data sets to the opening function. The code uses the MATLAB functions `peaks`, `membrane`, and `sinc`.

1. Display the opening function in the M-file editor. If the GUI M-file, `simple_gui.m`, is not already open in your editor, open it by selecting M-file Editor from the View menu. In the editor, click the function icon on the toolbar, then click **simple_gui_OpeningFcn** in the pop-up menu that displays.


Editor - D:\Work\Guide\Work\simple_gui.m
File Edit Text Go Cell Tools Debug Desktop Window Help
simple_gui
popupmenu1_Callback
popupmenu1_CreateFcn
pushbutton1_Callback
pushbutton5_Callback
pushbutton7_Callback
simple_gui_OpeningFcn
simple_gui_OutputFcn
function varargout = simple_gui(v
% SIMPLE_GUI M-file for simple_gu
% SIMPLE_GUI, by itself, cre
% singleton*.
% H = SIMPLE_GUI returns the
% the existing singleton*.
% SIMPLE_GUI('CALLBACK',hObject,eventData,handles,...)
% function named CALLBACK in SIMPLE_GUI.M with the giv
% SIMPLE_GUI('Property','Value',...) creates a new SIM
% existing singleton*. Starting from the left, proper
% applied to the GUI before simple_gui_OpeningFunction
% unrecognized property name or invalid value makes pr
% stop. All inputs are passed to simple_gui_OpeningFc
% *See GUI Options on GUIDE's Tools menu. Choose "GUI
simple_gui / simple_gui_OpeningFcn Ln 48 Col 1 OVR

The cursor moves to the opening function which already contains this code:

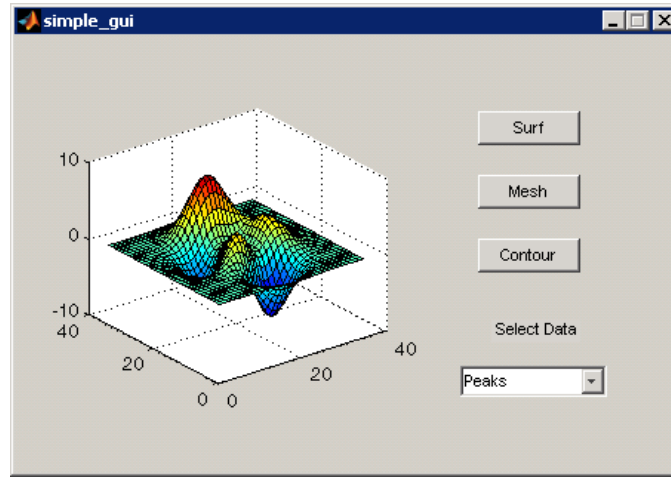
```
% --- Executes just before simple_gui is made visible.
function simple_gui_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to simple_gui (see VARARGIN)

% Choose default command line output for simple_gui
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes simple_gui wait for user response (see UIRESUME)
% uiwait(handles.figure1);
```

2 Create data for the GUI to plot by adding the following code to the opening function immediately after the comment that begins `% varargin`...

```
% Create the data to plot.
handles.peaks=peaks(35);
handles.membrane=membrane;
[x,y] = meshgrid(-8:.5:8);
r = sqrt(x.^2+y.^2) + eps;
sinc = sin(r)./r;
handles.sinc = sinc;
% Set the current data value.
handles.current_data = handles.peaks;
surf(handles.current_data)
```

The first six executable lines create the data using the MATLAB functions `peaks`, `membrane` and `sinc`. They store the data in the `handles` structure, which is passed as an argument to all callbacks. Callbacks for the push buttons can retrieve the data from the `handles` structure.

The last two lines create a current data value and set it to peaks, and then display the surf plot for peaks. The following figure shows how the GUI now looks when it first displays.

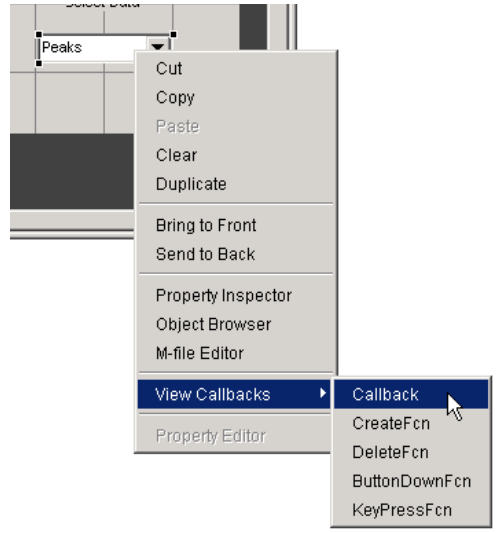


Programming the Pop-Up Menu

The pop-up menu enables users to select the data to plot. When a user selects one of the three plots, MATLAB sets the pop-up menu `Value` property to the

index of the selected string. The pop-up menu callback reads the pop-up menu `Value` property to determine what item is currently displayed and sets `handles.current_data` accordingly.

1 Display the pop-up menu callback in the M-file editor. Right click the pop-up menu component in the Layout Editor to display a context menu. From that menu, select **View Callbacks -> Callback**.



The GUI M-file opens in the editor if it is not already open, and the cursor moves to the pop-menu callback which already contains this code:

```
% --- Executes on selection change in popupmenu1.
function popupmenu1_Callback(hObject, eventdata, handles)
% hObject    handle to popupmenu1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

2 Add the following code to the `popupmenu1_Callback` after the comment that begins `% handles`...

This code first retrieves two pop-up menu properties:

- `String` – a cell array that contains the menu contents
- `Value` – the index into the menu contents of the selected data set.

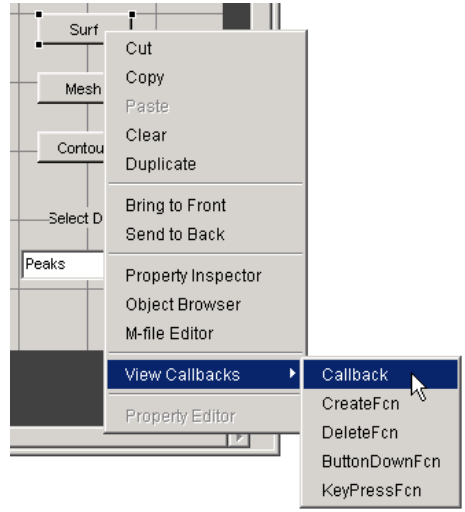
It then uses a `switch` statement to make the selected data set the current data. The last statement saves the changes to the `handles` structure.

```
% Determine the selected data set.
str = get(hObject, 'String');
val = get(hObject,'Value');
% Set current data to the selected data set.
switch str{val};
case 'Peaks' % User selects peaks
   handles.current_data = handles.peaks;
case 'Membrane' % User selects membrane
   handles.current_data = handles.membrane;
case 'Sinc' % User selects sinc
   handles.current_data = handles.sinc;
end
% Save the handles structure.
guidata(hObject,handles)
```

Programming the Push Buttons

Each of the push buttons creates a different type of plot using the data specified by the current selection in the pop-up menu. The push button callbacks get data from the `handles` structure and then plot it.

1. Display the **Surf** push button callback in the M-file editor. Right click the **Surf** pushbutton in the Layout Editor to display a context menu. From that menu select **View Callbacks -> Callback**.



The GUI M-file opens in the editor if it is not already open, and the cursor moves to the **Surf** push button callback which already contains this code:

```
% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

2 Add the following code to the callback immediately after the comment that begins `% handles...`

```
% Display surf plot of the currently selected data.
surf(handles.current_data);
```

3 Repeat steps **1** and **2** to add similar code to the **Mesh** and **Contour** push button callbacks.

- Add this code to the **Mesh** push button callback, `pushbutton2_Callback`:

  ```
  % Display mesh plot of the currently selected data.
  mesh(handles.current_data);
  ```

- Add this code to the **Contour** push button callback, `pushbutton3_Callback`:

```
% Display contour plot of the currently selected data.
contour(handles.current_data);
```

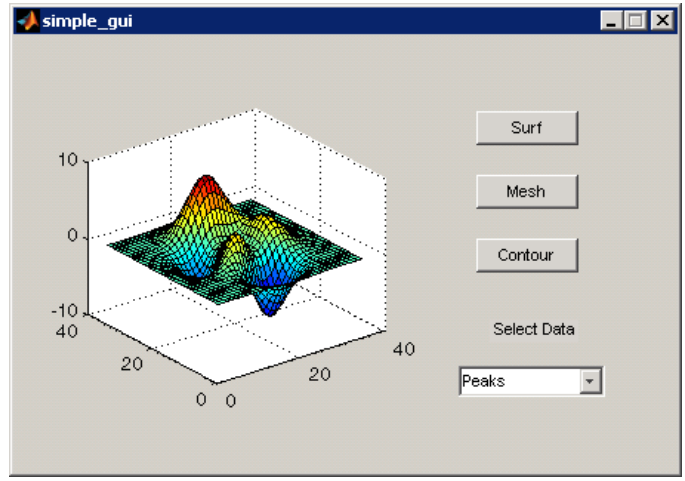
4 Save the M-file by selecting **Save** from the **File** menu.

Your GUI is ready to run. The next topic, “Running the GUI” on page 2-26, tells you how to do that.

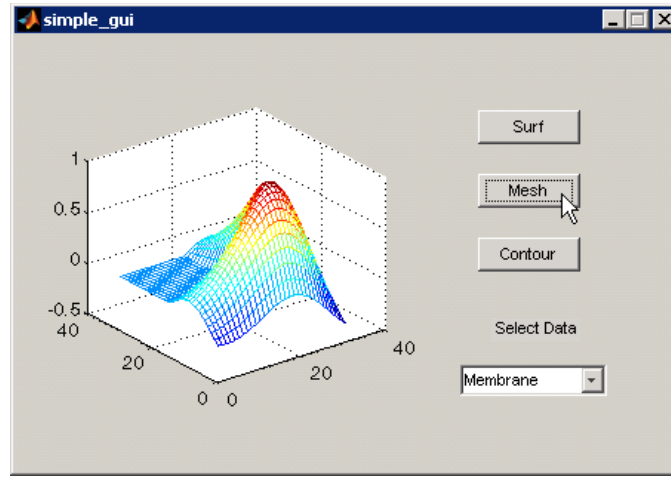
Running the GUI

In the previous topic, you programmed the pop-up menu and the push buttons. You also created data for them to use and initialized the display. Now you can run your GUI and see how it works.

1. Run your GUI by selecting **Run** from the Layout Editor **Tools** menu. If the GUI is on your MATLAB path or in your current directory, you can also run it by typing its name, `simple_gui`, at the prompt. The GUI looks like this when it first displays:



2. In the pop-up menu, select **Membrane**, then click the **Mesh** button. The GUI displays a mesh plot of the MATLAB logo.



3 Try some of the other combinations before closing the GUI.

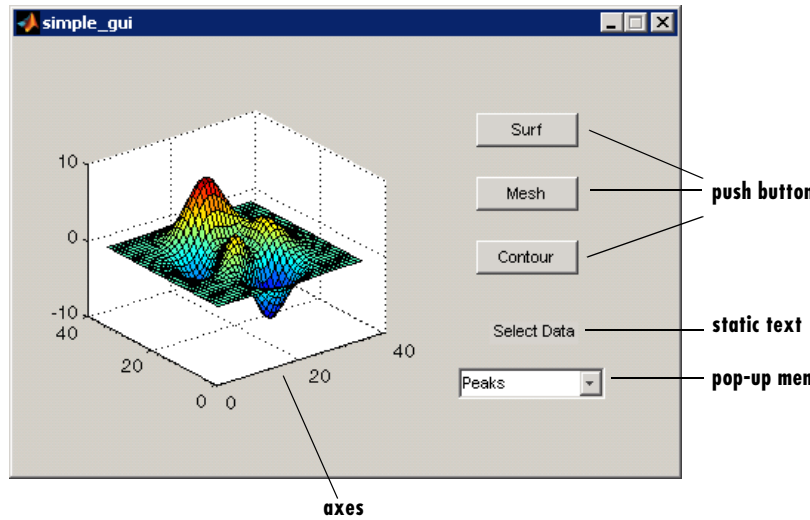
3

Creating a GUI Programmatically

The Example: Simple GUI (p. 3-2)	Describes the example to be constructed.
Function Summary (p. 3-4)	Lists the functions that are used in the construction of the example.
Creating a GUI M-File (p. 3-6)	Creates the file that holds the GUI script and adds `help` comments to the file.
Laying Out a Simple GUI (p. 3-7)	Creates the figure and adds the components.
Initializing the GUI (p. 3-11)	Performs various initialization chores and generates the data to plot
Programming the GUI (p. 3-14)	Adds code for each component to the GUI M-file to make the GUI work.
Running the Final GUI (p. 3-17)	Runs the final GUI and demonstrates how the components work together.

The Example: Simple GUI

This chapter shows you how to write a script that creates the example graphical user interface (GUI) shown in the following figure.



The GUI contains:

- An axes
- A pop-up menu listing three different data sets that correspond to MATLAB functions: `peaks`, `membrane`, and `sinc`
- A static text to label the pop-up menu
- Three push buttons, each od which provides a different kind of plot: surface, mesh, and contour

To use the GUI, the user selects a data set from the pop-up menu, then clicks one of the plot-type push buttons. Clicking the button triggers the execution of a callback that plots the selected data in the axes.

The next topic, “Function Summary” on page 3-4, summarizes the functions used to create this example GUI.

Subsequent topics guide you through the process of creating the GUI. This process begins with “Creating a GUI M-File” on page 3-6. We recommend that you create the GUI for yourself.

Function Summary

MATLAB provides a suite of functions for creating GUIs. This topic introduces you to the functions you need to create the example GUI shown in the following figure.

Functions Used to Create the Simple GUI

Function	Description
`align`	Align user interface controls and axes.
`axes`	Create axes objects.
`figure`	Create figure objects. A GUI is a figure object.
`movegui`	Move GUI figure to specified location on screen.
`uicontrol`	Create user interface control objects, such as a push buttons, static text, and pop-up menus.

Other MATLAB Functions Used in the GUI

Function	Description
`contour`	Contour graph of a matrix
`eps`	Floating point relative accuracy.
`get`	Query object properties.
`membrane`	Generates the MATLAB logo.
`mesh`	Mesh plots.
`meshgrid`	Generate X and Y arrays for 3-D plots.
`peaks`	Example function of two variables.
`set`	Set object properties.

Function	Description
`sin`	Sine; result in radians.
`sqrt`	Square root.
`surf`	3-D shaded surface plot.

Creating a GUI M-File

Start by creating an M-file for the example GUI.

1. At the MATLAB prompt, type `edit`. MATLAB opens the editor.

2. Type or copy the following statement into the editor. This function statement is the first line in the file.

   ```
   function simple_gui
   ```

3. Add these comments to the M-file following the function statement. They are displayed at the command line in response to the `help` command. They must be followed by a blank line.

   ```
   % SIMPLE_GUI Select a data set from the pop-up menu, then
   % click one of the plot-type push buttons. Clicking the button
   % plots the selected data in the axes.
   ```
   (Leave a blank line here)

4. Add an `end` statement at the end of the file. This `end` statement matches the `function` statement. Your file now looks like this.

   ```
   function simple_gui_test
   % SIMPLE_GUI Select a data set from the pop-up menu, then
   % click one of the plot-type push buttons. Clicking the button
   % plots the selected data in the axes.

   end
   ```

Note You need the `end` statement because the example is written using nested functions. For information about using nested functions, see Nested Functions in the MATLAB documentation.

---

5. Save the file in your current directory or at a location that is on your MATLAB path.

The next section, "Laying Out a Simple GUI" on page 3-7, shows you how to add components to your GUI.

Laying Out a Simple GUI

This topic illustrates the basic steps for laying out a GUI.

- "Creating the Figure" on page 3-7
- "Adding the Components" on page 3-7

Creating the Figure

In MATLAB, a GUI is a figure. This first step creates the figure and positions it on the screen. It also makes the GUI invisible so that the GUI user cannot see the components being added or initialized. When the GUI has all its components and is initialized, the example makes it visible.

```
%  Initialize and hide the GUI as it is being constructed.
f = figure('Visible','off','Position',[360,500,450,285]);
```

The call to the `figure` function uses two property/value pairs. The `Position` property is a 4-element vector that specifies the location of the GUI on the screen and its size: [distance from left, distance from bottom, height, width]. Default units are pixels.

The next topic, "Adding the Components" on page 3-7, shows you how to add the push buttons, axes, and other components to the GUI

Adding the Components

The example GUI has six components: 3 push buttons, 1 static text, 1 pop-up menu, and 1 axes. Start by writing statements that add these components to the GUI. Create the push buttons, static text, and pop-up menu with the `uicontrol` function. Use the `axes` function to create the axes.

1 Add the three push buttons to your GUI by adding these statements to your M-file following the call to `figure`.

```
% Construct the components.
hsurf    = uicontrol('Style','pushbutton',...
             'String','Surf','Position',[315,220,70,25]);
hmesh    = uicontrol('Style','pushbutton',...
             'String','Mesh','Position',[315,180,70,25]);
hcontour = uicontrol('Style','pushbutton',...
             'String','Countour','Position',[315,135,70,25]);
```

These statements use the `uicontrol` function to create the push buttons. Each statement uses a series of property/value pairs to define a push button.

Property	Description
`Style`	In the example, `pushbutton` specifies the component as a push button.
`String`	Specifies the label that appears on each push button. Here, there are three types of plots: `Surf`, `Mesh`, `Contour`.
`Position`	Uses a 4-element vector to specify the location of each push button within the GUI and its size: [distance from left, distance from bottom, height, width]. Default units for push buttons are pixels.

Each call returns the handle of the component that is created.

2 Add the pop-up menu and its label to your GUI by adding these statements to the M-file following the push button definitions.

```
hpopup = uicontrol('Style','popupmenu',...
           'String',{'Peaks','Membrane','Sinc'},...
           'Position',[300,50,100,25]);
htext  = uicontrol('Style','text','String','Select Data',...
           'Position',[325,90,60,15]);
```

For the pop-up menu, the `String` property uses a cell array to specify the three items in the pop-up menu: `Peaks`, `Membrane`, `Sinc`. The static text component serves as a label for the pop-up menu. Its `String` property tells the GUI user to `Select Data`. Default units for these components are pixels.

3 Add the axes to the GUI by adding this statement to the M-file. Set the `Units` property to pixels so that it has the same units as the other components.

```
ha = axes('Units','pixels','Position',[50,60,200,185]);
```

4 Align all components except the axes along their centers with the following statement. Add it to the M-file following all the component definitions.

```
align([hsurf,hmesh,hcontour,htext,hpopup],'Center','None');
```

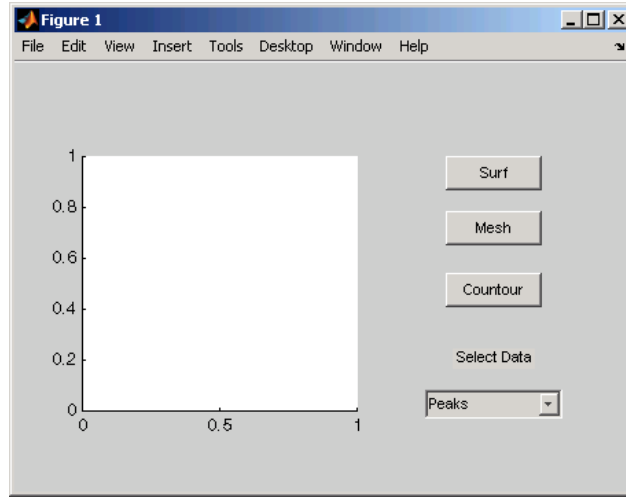
5 Make your GUI visible by adding this command following the `align` command.

```
set(f,'Visible','on')
```

6 This is what your M-file should now look like:

```
function simple_gui
% SIMPLE_GUI Select a data set from the pop-up menu, then
% click one of the plot-type push buttons. Clicking the button
% plots the selected data in the axes.

   %  Create and hide the GUI as it is being constructed.
   f = figure('Visible','off','Position',[360,500,450,285]);

   %  Construct the components
   hsurf = uicontrol('Style','pushbutton','String','Surf',...
           'Position',[315,220,70,25]);
   hmesh = uicontrol('Style','pushbutton','String','Mesh',...
           'Position',[315,180,70,25]);
   hcontour = uicontrol('Style','pushbutton',...
           'String','Countour',...
           'Position',[315,135,70,25]);
   htext = uicontrol('Style','text','String','Select Data',...
           'Position',[325,90,60,15]);
   hpopup = uicontrol('Style','popupmenu',...
           'String',{'Peaks','Membrane','Sinc'},...
           'Position',[300,50,100,25]);
   ha = axes('Units','Pixels','Position',[50,60,200,185]);
   align([hsurf,hmesh,hcontour,htext,hpopup],'Center','None');

   %Make the GUI visible.
   set(f,'Visible','on')

end
```

7 Run your script by typing `simple_gui` at the command line. This is what your GUI now looks like. Note that you can select a data set in the pop-up menu and click the push buttons. But nothing happens. This is because there is no code in the M-file to service the pop-up menu and the buttons.



8 Type `help simple_gui` at the command line. MATLAB displays this help text.

```
help simple_gui
  SIMPLE_GUI Select a data set from the pop-up menu, then
  click one of the plot-type push buttons. Clicking the button
  plots the selected data in the axes.
```

The next topic, “Initializing the GUI” on page 3-11, shows you how to initialize the GUI.

Initializing the GUI

When you make the GUI visible, it should be initialized so that it is ready for the user. This topic shows you how to:

- Make the GUI behave properly when it is resized by changing the component and figure units to `normalized`.

  This causes the components to resize when the GUI is resized. Normalized units map the lower left corner of the figure window to `(0,0)` and the upper right corner to `(1.0, 1.0)`.
- Generate the data to plot.

  The example needs three sets of data: `peaks_data`, `membrane_data`, and `sinc_data`. Each set corresponds to one of the items in the pop-up menu.
- Create an initial plot in the axes
- Assign the GUI a name that appears in the window title
- Move the GUI to the center of the screen
- Make the GUI visible

1 Replace this code in your M-file:

```
% Make the GUI visible.
set(f,'Visible','on')
```

with this code:

```
% Initialize the GUI.
% Change units to normalized so components resize automatically.
set([f,hsurf,hmesh,hcontour,htext,hpopup],'Units','normalized');
% Generate the data to plot.
peaks_data = peaks(35);
membrane_data = membrane;
[x,y] = meshgrid(-8:.5:8);
r = sqrt(x.^2+y.^2) + eps;
sinc_data = sin(r)./r;
% Create a plot in the axes.
current_data = peaks_data;
surf(current_data);
% Assign the GUI a name to appear in the window title.
set(f,'Name','Simple GUI')
```

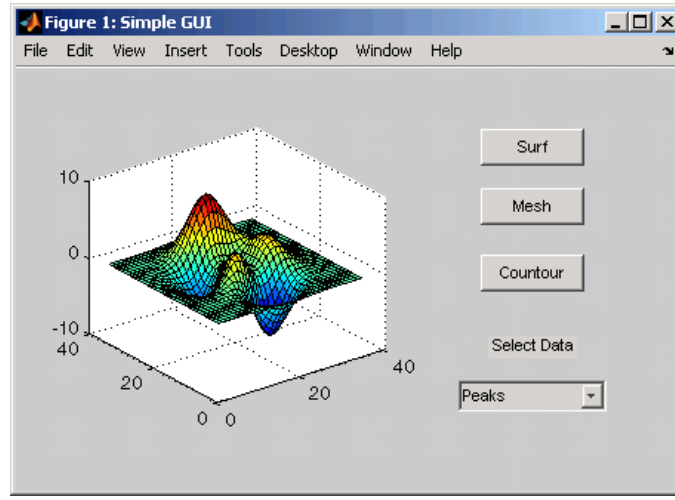
```
% Move the GUI to the center of the screen.
movegui(f,'center')
% Make the GUI visible.
set(f,'Visible','on');
```

2 This is what your M-file should now look like:

```
function simple_gui
% SIMPLE_GUI Select a data set from the pop-up menu, then
% click one of the plot-type push buttons. Clicking the button
% plots the selected data in the axes.

   % Create and hide the GUI figure as it is being constructed.
   f = figure('Visible','off','Position',[360,500,450,285]);

   % Construct the components
   hsurf = uicontrol('Style','pushbutton','String','Surf',...
           'Position',[315,220,70,25]);
   hmesh = uicontrol('Style','pushbutton','String','Mesh',...
           'Position',[315,180,70,25]);
   hcontour = uicontrol('Style','pushbutton',...
           'String','Countour',...
           'Position',[315,135,70,25]);
   htext = uicontrol('Style','text','String','Select Data',...
           'Position',[325,90,60,15]);
   hpopup = uicontrol('Style','popupmenu',...
           'String',{'Peaks','Membrane','Sinc'},...
           'Position',[300,50,100,25]);
   ha = axes('Units','Pixels','Position',[50,60,200,185]);
   align([hsurf,hmesh,hcontour,htext,hpopup],'Center','None');

   % Create the data to plot
   peaks_data = peaks(35);
   membrane_data = membrane;
   [x,y] = meshgrid(-8:.5:8);
   r = sqrt(x.^2+y.^2) + eps;
   sinc_data = sin(r)./r;

   % Initialize the GUI.
   % Change units to normalized so components resize
   % automatically.
```

```
    set([f,hsurf,hmesh,hcontour,htext,hpopup],...
        'Units','normalized');
    %Create a plot in the axes.
    current_data = peaks_data;
    surf(current_data);
    % Assign the GUI a name to appear in the window title.
    set(f,'Name','Simple GUI')
    % Move the GUI to the center of the screen.
    movegui(f,'center')
    % Make the GUI visible.
    set(f,'Visible','on');

end
```

3 Run your script by typing `simple_gui` at the command line. This is what your GUI should now look like.



The next topic, “Programming the GUI” on page 3-14, shows you how to program the push buttons and pop-up menu so you can interactively generate different plots in the axes.

Programming the GUI

You use callbacks to program the GUI components. Callbacks are the functions that execute in response to user-generated events such as a mouse click.

This topic shows you how to write callbacks for the simple GUI, then shows you how to associate each callback with its component.

- "Programming the Pop-Up Menu" on page 3-14
- "Programming the Push Buttons" on page 3-15
- "Associating Callbacks with Their Components" on page 3-15

Programming the Pop-Up Menu

The pop-up menu enables users to select the data to plot. When a GUI user selects one of the three data sets, MATLAB sets the pop-up menu `Value` property to the index of the selected string. The pop-up menu callback reads the pop-up menu `Value` property to determine which item is currently displayed and sets `current_data` accordingly.

Add the following callback to your file following the initialization code and before the final `end` statement.

```
%  Pop-up menu callback. Read the pop-up menu Value property
%  to determine which item is currently displayed and make it
%  the current data. This callback automatically has access to
%  current_data because this function is nested at a lower level.

   function popup_menu_Callback(source,eventdata)
      % Determine the selected data set.
      str = get(source, 'String');
      val = get(source,'Value');
      % Set current data to the selected data set.
      switch str{val};
      case 'Peaks' % User selects Peaks
         current_data = peaks_data;
      case 'Membrane' % User selects Membrane
         current_data = membrane_data;
      case 'Sinc' % User selects Sinc
         current_data = sinc_data;
      end
```

```
    end
```

The next topic, "Programming the Push Buttons" on page 3-15, shows you how to write callbacks for the three push buttons.

Programming the Push Buttons

Each of the three push buttons creates a different type of plot using the data specified by the current selection in the pop-up menu. The push button callbacks plot the data in `current_data`. They automatically have access to `current_data` because they are nested at a lower level.

Add the following callbacks to your file following the pop-up menu callback and before the final `end` statement.

```
% Push button callbacks. Each callback plots current_data in the
% specified plot type.

function surfbutton_Callback(source,eventdata)
% Display surf plot of the currently selected data.
     surf(current_data);
end

function meshbutton_Callback(source,eventdata)
% Display mesh plot of the currently selected data.
     mesh(current_data);
end

function contourbutton_Callback(source,eventdata)
% Display contour plot of the currently selected data.
     contour(current_data);
end
```

The next topic shows you how to associate each of the callbacks you've created with its specific component.

Associating Callbacks with Their Components

When the GUI user selects a data set from the pop-up menu or clicks one of the push buttons, MATLAB executes the callback associated with that particular event. But how does MATLAB know which callback to execute? You must use

each component's `Callback` property to specify the name of the callback with which it is associated.

1. To the `uicontrol` statement that defines the **Surf** push button, add the property/value pair

   ```
   'Callback',{@surfbutton_Callback}
   ```

   so that the statement looks like this:

   ```
   hsurf = uicontrol('Style','pushbutton','String','Surf',...
           'Position',[315,220,70,25],...
           'Callback',{@surfbutton_Callback});
   ```

   `Callback` is the name of the property. `surfbutton_Callback` is the name of the callback that services the **Surf** push button.

2. Similarly, to the `uicontrol` statement that defines the **Mesh** push button, add the property/value pair

   ```
   'Callback',{@meshbutton_Callback}
   ```

3. To the `uicontrol` statement that defines the **Contour** push button, add the property/value pair

   ```
   'Callback',{@contourbutton_Callback}
   ```

4. To the `uicontrol` statement that defines the pop-up menu, add the property/value pair

   ```
   'Callback',{@popup_menu_Callback}
   ```

The next topic, "Running the Final GUI" on page 3-17, shows the final M-file and runs the GUI.

Running the Final GUI

You have created the simple GUI. This topic shows you the final M-file and runs the GUI.

- "The Final M-File" on page 3-17
- "Running the GUI" on page 3-19

The Final M-File

This is what your final M-file should now look like:

```
function simple_gui
% SIMPLE_GUI Select a data set from the pop-up menu, then
% click one of the plot-type push buttons. Clicking the button
% plots the selected data in the axes.

   %  Create and then hide the GUI as it is being constructed.
   f = figure('Visible','off','Position',[360,500,450,285]);

   %  Construct the components
   hsurf = uicontrol('Style','pushbutton','String','Surf',...
          'Position',[315,220,70,25],...
          'Callback',{@surfbutton_Callback});
   hmesh = uicontrol('Style','pushbutton','String','Mesh',...
          'Position',[315,180,70,25],...
          'Callback',{@meshbutton_Callback});
   hcontour = uicontrol('Style','pushbutton',...
          'String','Countour',...
          'Position',[315,135,70,25],...
          'Callback',{@contourbutton_Callback});
   htext = uicontrol('Style','text','String','Select Data',...
          'Position',[325,90,60,15]);
   hpopup = uicontrol('Style','popupmenu',...
          'String',{'Peaks','Membrane','Sinc'},...
          'Position',[300,50,100,25],...
          'Callback',{@popup_menu_Callback});
   ha = axes('Units','Pixels','Position',[50,60,200,185]);
   align([hsurf,hmesh,hcontour,htext,hpopup],'Center','None');
```

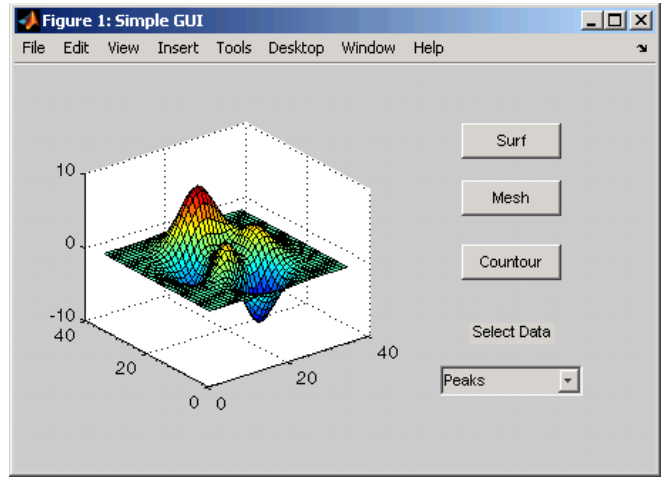
```
% Create the data to plot
peaks_data = peaks(35);
membrane_data = membrane;
[x,y] = meshgrid(-8:.5:8);
r = sqrt(x.^2+y.^2) + eps;
sinc_data = sin(r)./r;

% Initialize the GUI.
%Change units to normalized so components resize automatically.
set([f,ha,hsurf,hmesh,hcontour,htext,hpopup],...
'Units','normalized');
%Create a plot in the axes.
current_data = peaks_data;
surf(current_data);
% Assign the GUI a name to appear in the window title.
set(f,'Name','Simple GUI')
% Move the GUI to the center of the screen.
movegui(f,'center')
% Make the GUI visible.
set(f,'Visible','on');

%  Callbacks for simple_gui. These callbacks automatically have
%  access to component handles and initialized data because
%  they are nested at a lower level.

%  Pop-up menu callback. Read the pop-up menu Value property
%  to determine which item is currently displayed and make it
%  the current data.
   function popup_menu_Callback(source,eventdata)
      % Determine the selected data set.
      str = get(source, 'String');
      val = get(source,'Value');
      % Set current data to the selected data set.
      switch str{val};
      case 'Peaks' % User selects Peaks
         current_data = peaks_data;
      case 'Membrane' % User selects Membrane
         current_data = membrane_data;
      case 'Sinc' % User selects Sinc
         current_data = sinc_data;
```

```
            end
        end

    % Push button callbacks. Each callback plots current_data in the
     % specified plot type.

     function surfbutton_Callback(source,eventdata)
     % Display surf plot of the currently selected data.
        surf(current_data);
     end

     function meshbutton_Callback(source,eventdata)
     % Display mesh plot of the currently selected data.
        mesh(current_data);
     end

     function contourbutton_Callback(source,eventdata)
     % Display contour plot of the currently selected data.
        contour(current_data);
     end

end
```

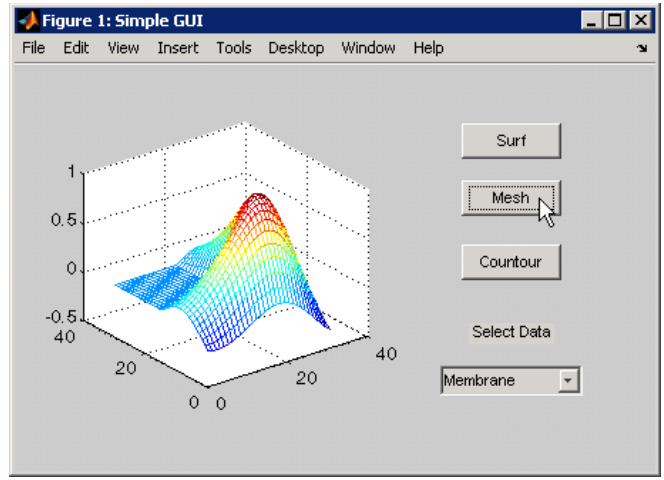
Running the GUI

1 Run the simple GUI by typing the name of the M-file at the command line.

```
simple_gui
```



2 In the pop-up menu, select **Membrane**, then click the **Mesh** button. The GUI displays a mesh plot of the MATLAB logo.



3 Try some of the other combinations before closing the GUI.

Creating GUIs with GUIDE

These sections describe how to create graphical user interfaces (GUIs) using GUIDE, the MATLAB® graphical user interface development environment.

What Is GUIDE? (p. 4-1)	Introduces GUIDE.
Preferences and Options (p. 5-1)	Describes briefly the available MATLAB preferences and GUI options.
Laying Out a GUI (p. 6-1)	Shows you how to start GUIDE and from there how to populate the GUI and create menus. Provides guidance in designing a GUI for cross-platform compatibility.
Saving and Running the GUI (p. 7-1)	Describes the files used to store the GUI. Steps you through the process for saving a GUI, and lists the different ways in which you can activate a GUI.
Programming the GUI (p. 8-1)	Explains how user-written callback routines control GUI behavior. Shows you how to associate callbacks with specific components and explains callback syntax and arguments. Provides simple programming examples for each kind of component.
Managing Application-Defined Data (p. 9-1)	Explains the mechanisms for managing application-defined data and explains how to share data among a GUIs callbacks.
Examples of GUIDE GUIs (p. 10-1)	Illustrates techniques for programming various behaviors.

Note Following the release of MATLAB version 7.2, Creating Graphical User Interfaces will be further updated and expanded. The HTML and PDF versions of this document will be updated on The MathWorks Web site some time after the release. Check this page and the title page of the PDF document to determine if they have been updated.

4

What Is GUIDE?

What Is GUIDE? (p. 4-2)	Introduces GUIDE, the MATLAB graphical user interface development environment.
GUIDE Tools Summary (p. 4-3)	Introduces the various tools that comprise GUIDE.

What Is GUIDE?

GUIDE, the MATLAB Graphical User Interface development environment, provides a set of tools for creating graphical user interfaces (GUIs). These tools simplify the process of laying out and programming GUIs.

Laying Out a GUI

Using the GUIDE Layout Editor, you can populate a GUI by clicking and dragging GUI components — such as axes, panels, buttons, text fields, sliders, and so on — into the layout area. You can also create menus and context menus for the GUI. From the Layout Editor, you can size the GUI, modify component look and feel, align components, set tab order, view a hierarchical list of the component objects, and set GUI options.

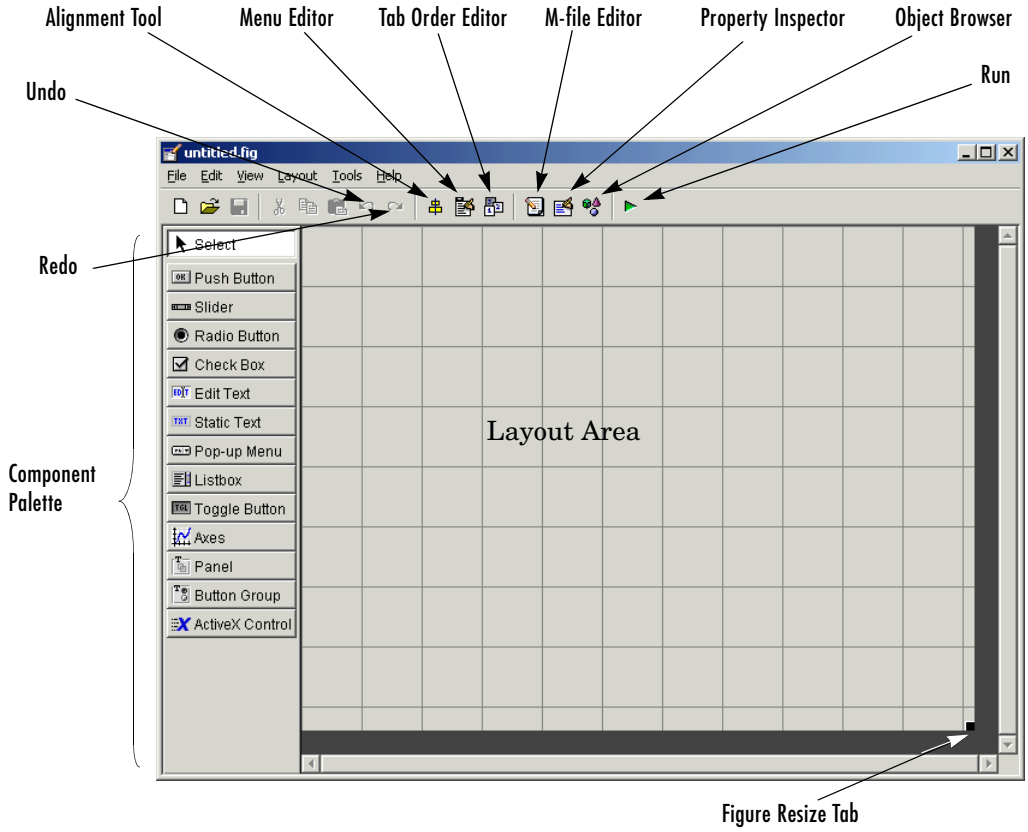
Programming the GUI

GUIDE automatically generates an M-file that controls how the GUI operates. This M-file provides code to initialize the GUI and contains a framework for the GUI callbacks — the routines that execute when a user clicks a GUI component. Using the M-file editor, you can add code to the callbacks to perform the functions you want them to.

Note MATLAB provides a selection of standard dialog boxes that you can create with a single function call. For information about these dialog boxes and the functions used to create them, see Predefined Dialog Boxes in the MATLAB documentation.

GUIDE Tools Summary

The GUIDE tools are available from the Layout Editor shown in the figure below. The tools are called out in the figure and described briefly below. Subsequent sections show you how to use them.



Tool	Use This Tool To:
Layout Editor	Select components from the Component Palette, at the left side of the Layout Editor, and arrange them in the Layout Area. See "Adding Components to the GUI" on page 6-14 for more information.
Figure Resize Tab	Set the size at which the GUI is initially displayed. See "Setting the GUI Size" on page 6-12, for more information.
Menu Editor	Create menus and context, i.e., pop-up, menus. See "Creating Menus" on page 6-48 for more information.
Align Objects	Align and distribute groups of components. Grids and rulers also enable you to align components on a grid with optional snap to grid. See "Aligning Components" on page 6-39 for more information.
Tab Order Editor	Set the tab and stacking order of the components in your layout. See "Setting Tab Order" on page 6-45 for more information.
Property Inspector	Set the properties of the components in your layout. It provides a list of all the properties you can set and displays their current values.
Object Browser	Display a hierarchical list of the objects in the figure. See "Viewing the Object Hierarchy" on page 6-62 for more information.

Tool	Use This Tool To:
Run	Save and run the current GUI, and display the M-file associated with the GUI in your default editor. See Chapter 7, “Saving and Running the GUI” for information.
M-File Editor	Display, in your default editor, the M-file associated with the GUI. See “GUI Files: An Overview” on page 8-5 for more information.

You can also set preferences that apply to all GUIs at creation, and options that are GUI-specific. See Chapter 5, “Preferences and Options” for more information.

5

Preferences and Options

GUIDE Preferences (p. 5-2)	MATLAB preferences for the GUIDE Layout Editor.
GUI Options (p. 5-9)	GUIDE options for individual GUIs.

GUIDE Preferences

You can set preferences for the GUIDE Layout Editor by selecting **Preferences** from the MATLAB **File** menu. These preferences apply to all GUIs you create.

The preferences are found in different locations within the Preferences dialog box:

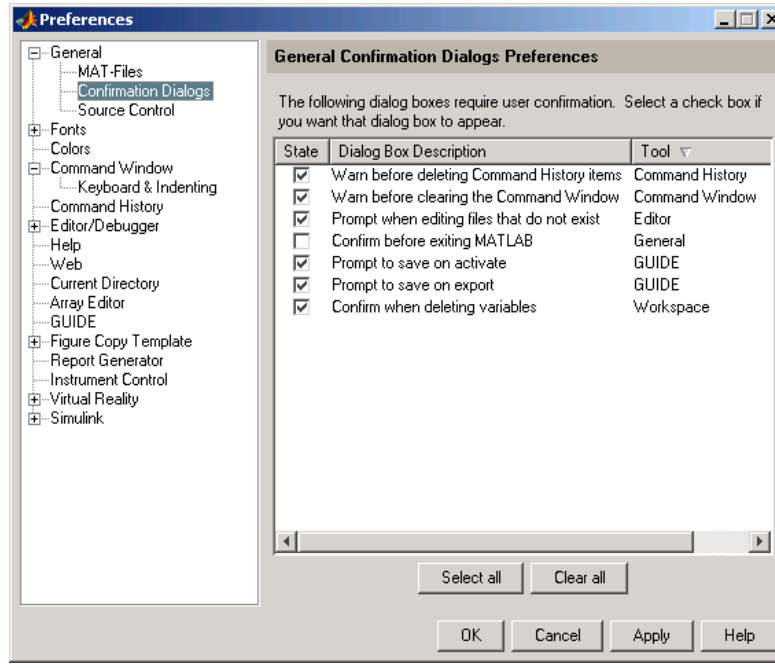
- "Confirmation Preferences" on page 5-2
- "Backward Compatibility Preference" on page 5-4
- "All Other Preferences" on page 5-5

Confirmation Preferences

The GUIDE Layout Editor provides two confirmation preferences. You can choose whether you want to display a confirmation dialog box when you:

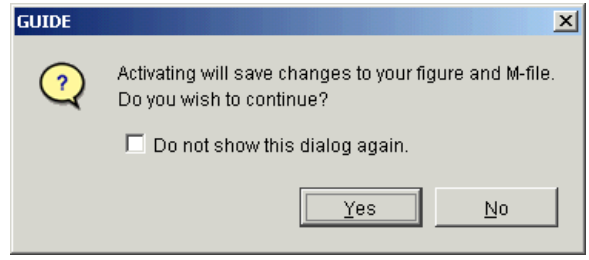
- Activate a GUI
- Export a GUI

In the MATLAB Preferences dialog box, click **General -> Confirmation Dialogs** to access the Layout Editor confirmation preferences. Look for the word `GUIDE` in the **Tool** column.



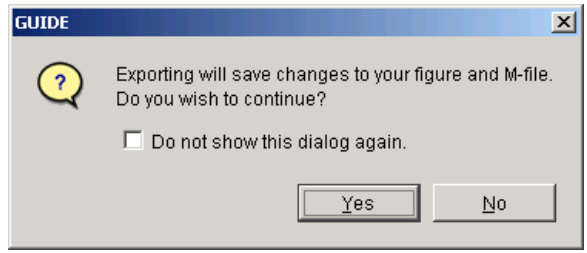
Prompt to Save on Activate

When you activate a GUI by clicking the Run button , a dialog box informs you of the impending save and lets you choose whether or not you want to continue.



Prompt to Save on Export

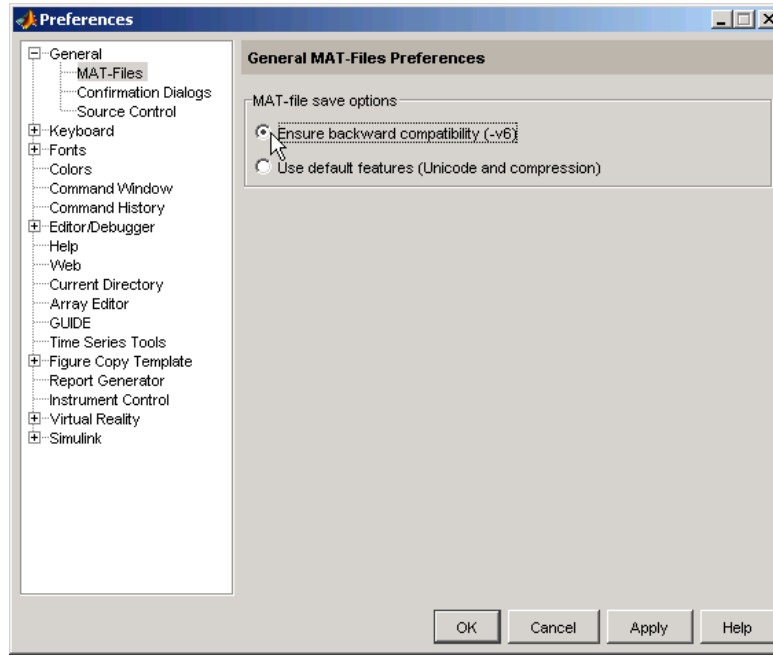
When you select **Export** from the Layout Editor **File** menu, a dialog box informs you of the impending save and lets you choose whether or not you want to continue.



Backward Compatibility Preference

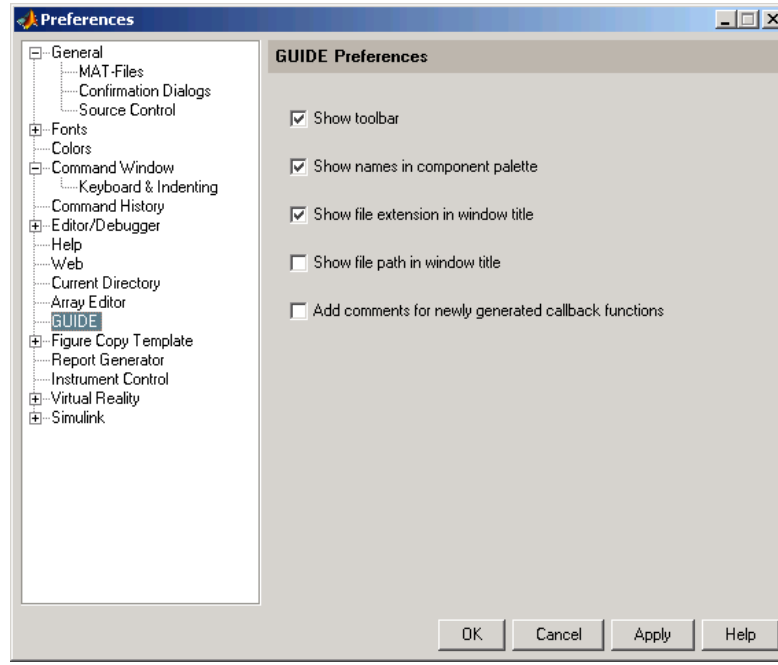
Ensure Backward Compatibility (-v6)

GUI FIG-files that are created or modified with MATLAB 7.0 or a later MATLAB version, are not automatically compatible with Version 6.5 and earlier versions. To make a FIG-file, which is a kind of MAT-file, backward compatible, you must check the MAT-file **Ensure backward compatibility (-v6)** preference in the **Preferences** dialog box under **General** -> **MAT-Files**. This is shown in the figure below.



All Other Preferences

The GUIDE Layout Editor provides several other preferences. In the MATLAB Preferences dialog box, click **GUIDE** to access the remaining Layout Editor preferences. They are shown in the following figure.

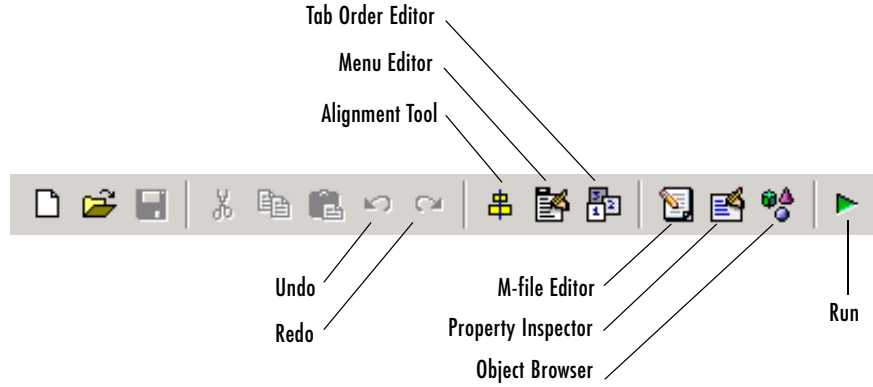


The following topics describe the preferences in this dialog:

- "Show Toolbar" on page 5-6
- "Show Names in Component Palette" on page 5-7
- "Show File Extension in Window Title" on page 5-8
- "Show File Path in Window Title" on page 5-8
- "Add Comments for Newly Generated Callback Functions" on page 5-8

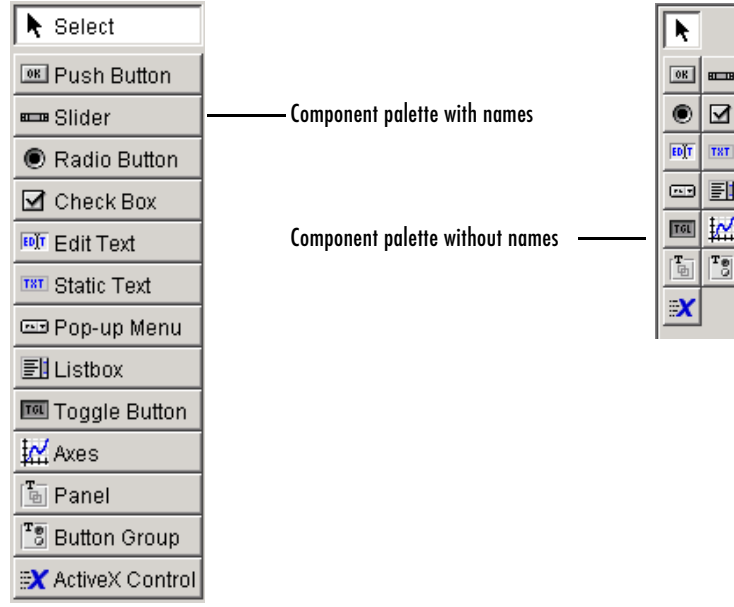
Show Toolbar

Displays the following toolbar in the Layout Editor window.



Show Names in Component Palette

Displays both icons and names in the component palette, as shown below. When unchecked, the icons alone are displayed in a two-column array.



Show File Extension in Window Title

Displays the filename with its file extension in the Layout Editor window title. If unchecked, only the filename is displayed.

Show File Path in Window Title

Displays the full file path in the Layout Editor window title. If unchecked, the file path is not displayed.

Add Comments for Newly Generated Callback Functions

When this preference is checked, GUIDE includes the comment lines shown in the following example to all callbacks that are added to the M-file. Some callbacks are added automatically because their associated components are part of the original GUDE template that you chose. Other commonly used callbacks are added automatically when you add components. You can also add callbacks explicitly by selecting them from **View Callbacks** on the **View** menu or on the component's context menu.

```
% --- Executes during object deletion, before destroying properties.
function figure1_DeleteFcn(hObject, eventdata, handles)
% hObject    handle to figure1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

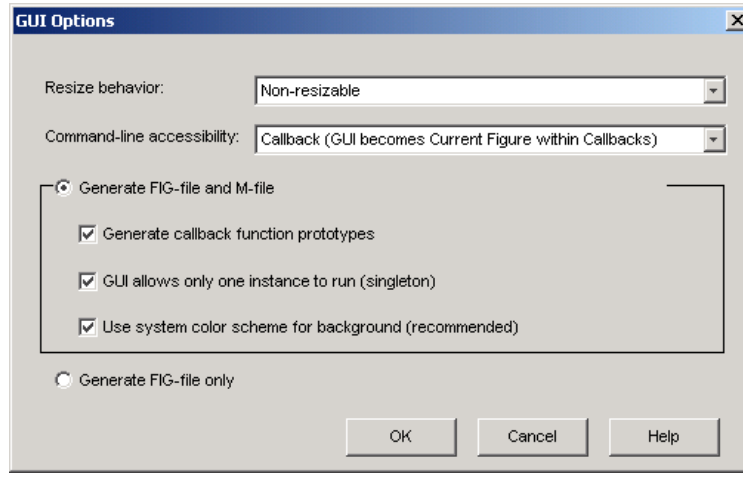
If this preference is unchecked, GUIDE includes comments only for callbacks that are automatically included to support the original GUIDE template. No comments are included for any other callbacks that are added to the M-file.

See "Callback Syntax and Arguments" on page 8-11 for more information about callbacks and about the arguments described in the comments above.

GUI Options

After opening a new GUI template in the Layout Editor, but before saving the GUI, you can configure various behaviors for the GUI with the **GUI Options** dialog box. These options apply only to the specific GUI you are creating.

Access the dialog box by selecting **GUI Options** from the Layout Editor **Tools** menu.



The following sections describe the options in this dialog:

- "Resize Behavior" on page 5-9
- "Command-Line Accessibility" on page 5-10
- "Generate FIG-File and M-File" on page 5-11
  - "Generate Callback Function Prototypes" on page 5-11
  - "GUI Allows Only One Instance to Run (Singleton)" on page 5-12
  - "Using the System Background Colors" on page 5-12
- "Generate FIG-File Only" on page 5-13

Resize Behavior

You can control whether users can resize the figure window containing your GUI and how MATLAB handles resizing. GUIDE provides three options:

- **Non-resizable** — Users cannot change the window size (default).
- **Proportional** — MATLAB automatically rescales the components in the GUI in proportion to the new figure window size.
- **Other (Use ResizeFcn)** — Program the GUI to behave in a certain way when users resize the figure window.

The first two approaches simply set properties appropriately and require no other action. **Other (Use ResizeFcn)** requires you to write a callback routine that recalculates sizes and positions of the components based on the new figure size.

Command-Line Accessibility

You can restrict access to a GUI figure from the command line by using the GUIDE **Command-line accessibility** options.

Many commands, such as `plot`, alter the current figure, i.e., the figure specified by the root `CurrentFigure` property. The current figure is usually the figure that is most recently created or clicked in. However, a figure can also become the current figure with the statement

```
figure(h)
```

or by setting the `CurrentFigure` property to the figure's handle.

For a GUI created in GUIDE set the **Command-line accessibility** options to prevent users from inadvertently changing the appearance of a GUI by entering commands at the command line, such as `plot`, that alter the current figure.

The following table briefly describes the four options for **Command-line accessibility**.

Option	Description
Callback (GUI becomes Current Figure within Callbacks)	The GUI can be accessed only from within a GUI callback. The GUI cannot be accessed from the command line or from an M-script. This is the default.
Off (GUI never becomes Current Figure)	The GUI can not be accessed from a callback, the command line, or an M-script, without the handle.
On (GUI may become Current Figure from Command Line)	The GUI can be accessed from a callback, from the command line, and from an M-script.
Other (Use settings from Property Inspector)	You control accessibility by setting the `HandleVisibility` and `IntegerHandle` properties from the Property Inspector.

Generate FIG-File and M-File

Select **Generate FIG-file and M-file** in the **GUI Options** dialog if you want GUIDE to create both the FIG-file and the GUI M-file (this is the default). Once you have selected this option, you can select any of the following items in the frame to configure the M-file:

- Generate callback function prototypes
- Application allows only one instance to run (singleton)
- Use system color scheme for background

See “GUI Files: An Overview” on page 8-5 for information about these files.

Generate Callback Function Prototypes

If you select **Generate callback function prototypes** in the **GUI Options** dialog, GUIDE adds templates for the most commonly used callbacks to the GUI M-file for any component you add to the GUI. You must then write the code for these callbacks.

GUIDE also adds a callback whenever you edit a callback routine from the Layout Editor's right-click context menu and when you add menus to the GUI using the Menu Editor.

See "Callback Syntax and Arguments" on page 8-11 for general information about callbacks.

Note This option is available only if you first select the **Generate FIG-file and M-File** option.

GUI Allows Only One Instance to Run (Singleton)

This option allows you to select between two behaviors for the GUI figure:

- Allow MATLAB to display only one instance of the GUI at a time.
- Allow MATLAB to display multiple instances of the GUI.

If you allow only one instance, MATLAB reuses the existing GUI figure whenever the command to run the GUI is issued. If a GUI already exists, MATLAB brings it to the foreground rather than creating a new figure.

If you clear this option, MATLAB creates a new GUI figure whenever you issue the command to run the GUI.

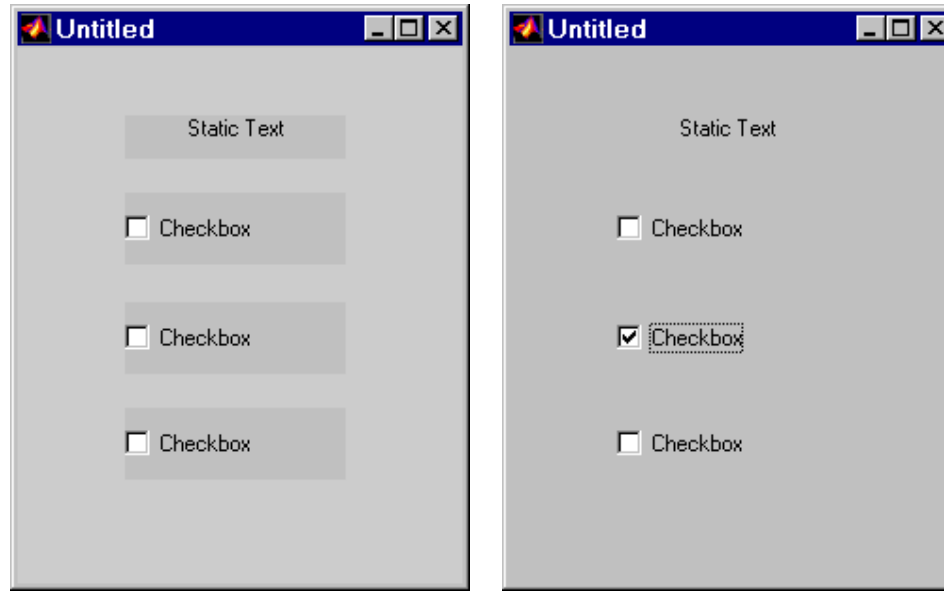
Note This option is available only if you first select the **Generate FIG-file and M-File** option.

Using the System Background Colors

The color used for GUI components varies on different computer systems. This option enables you to make the figure background color the same as the default uicontrol background color, which is system dependent.

If you select **Use system color scheme for background** (the default), GUIDE changes the figure background color to match the color of the GUI components.

The following figures illustrate the results with and without system color matching.



Without system color matching

With system color matching

Note This option is available only if you first select the **Generate FIG-file and M-File** option.

Generate FIG-File Only

The **Generate FIG-file only** option can be selected by GUIDE or by you, the user.

- GUIDE selects **Generate FIG-file only** as the default if, when you start GUIDE, you provide the name of a FIG-file or the handle of a figure for which a FIG-file exists. For example,

  ```
  guide('myfig.fig')
  ```

  The figure `myfig.fig` need not be generated using GUIDE. This mode provides limited editing capability and may be useful for GUIs generated in

MATLAB versions 5.3 and earlier. See the guide command for more information.

- If you select **Generate FIG-file only** GUIDE generates only a FIG-file when you save the GUI from the Layout Editor. You might do this if the layout is not yet complete, but you want to save your work without generating the M-file. You can redisplay the GUI in the Layout Editor using the `open` command.

  You can run a GUI that has no M-file only if you set the `Callback` property of each component in your GUI to a string that MATLAB can evaluate and perform the desired action. This string can be an expression or the name of an M-file.

  This option also allows you to create your own M-file. One that may use a completely different programming style than that provided by the GUIDE M-file.

6

Laying Out a GUI

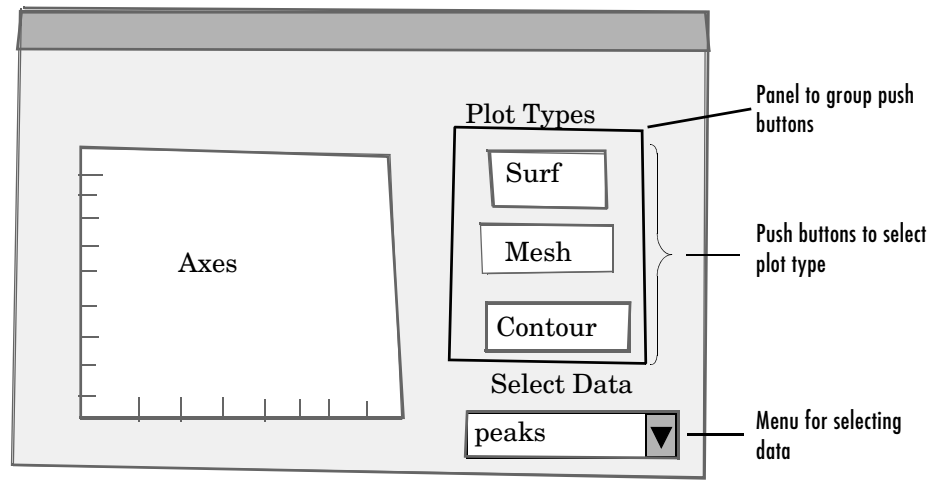
Designing a GUI (p. 6-2)	Things to think about when designing a GUI and references to other sources.
Starting GUIDE (p. 6-4)	Shows you all the ways to start GUIDE.
Selecting a GUI Template (p. 6-5)	Describes the templates from which you can choose when you create a new GUI.
Setting the GUI Size (p. 6-12)	Shows you how to set the size at which a GUI is initially displayed.
Adding Components to the GUI (p. 6-14)	Describes the process for adding components to a GUI, labeling them, and assigning identifiers to them. It also shows you how to move, copy, paste, duplicate, and resize components.
Aligning Components (p. 6-39)	Describes various approaches for aligning components.
Setting Tab Order (p. 6-45)	Explains tab order and shows you how to set it.
Creating Menus (p. 6-48)	Shows you how to create both menus that appear on the figure menu bar and context menus.
Creating Toolbars (p. 6-61)	Provides basic direction for adding toolbars to your GUI programmatically.
Viewing the Object Hierarchy (p. 6-62)	Describes use of the Object Browser to view the hierarchy of objects, including menus, in your GUI.
Designing for Cross-Platform Compatibility (p. 6-63)	Provides pointers for creating GUIs that behave more consistently when run on different platforms.

Designing a GUI

Before creating the actual GUI, it is important to decide what it is you want your GUI to do and how you want it to work. It is helpful to draw your GUI on paper and envision what the user sees and what actions the user takes.

Note MATLAB provides a selection of standard dialog boxes that you can create with a single function call. For information about these dialog boxes and the functions used to create them, see Predefined Dialog Boxes in the MATLAB documentation.

The GUI used in this example contains an axes that displays either a surface, mesh, or contour plot of data selected from the pop-up menu. The following picture shows a sketch that you might use as a starting point for the design.



A panel contains three push buttons that enable you to select the type of plot you want. The pop-up menu contains three strings — `peaks`, `membrane`, and `sinc`, which correspond to MATLAB functions. You can select the data to plot from this menu.

Many web sites and commercial publications such as the following provide guidelines for designing GUIs:

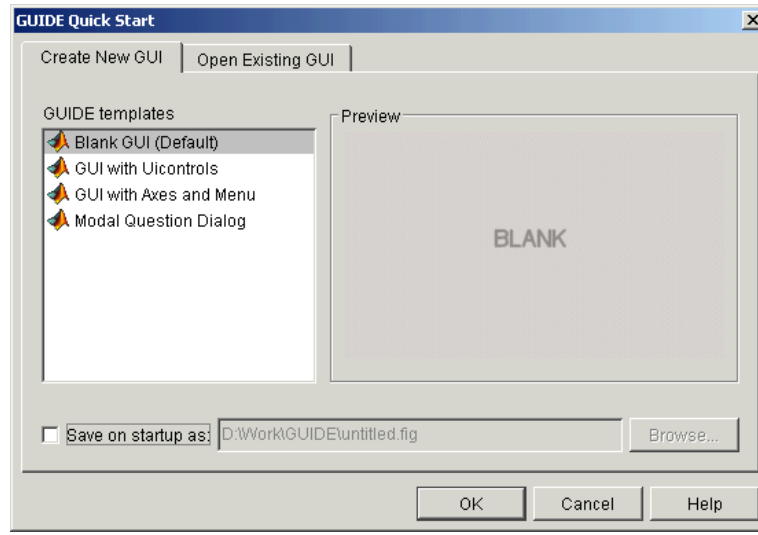
- AskTog – Essays on good design and a list of First Principles for good user interface design. The author, Tognazzini, is a well-respected user interface designer. `http://www.asktog.com/basics/firstPrinciples.html`.
- Galitz, Wilbert, O., *Essential Guide to User Interface Design*. Wiley, New York, NY, 2002.
- GUI Design Handbook – A detailed guide to the use of GUI controls. `http://www.fast-consulting.com/GUI%20Design%20Handbook/GDH_FRNTMTR.htm`.
- Johnson, J., *GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers*. Morgan Kaufmann, San Francisco, CA, 2000.
- Usability Glossary – An extensive glossary of terms related to GUI design, usability, and related topics. `http://www.usabilityfirst.com/glossary/main.cgi`.
- UsabilityNet – Covers design principles, user-centered design, and other usability and design-related topics. `http://www.usabilitynet.org/management/b_design.htm`.

Starting GUIDE

There are many ways to start GUIDE. You can start GUIDE:

- From the command line by typing `guide`.
- From the **Start** menu by selecting **MATLAB -> GUIDE (GUI Builder)**.
- From the **File** menu by selecting **New -> GUI**.
- From the MATLAB toolbar by selecting the GUIDE icon .

However you start GUIDE, it displays the **GUIDE Quick Start** dialog box shown in the following figure.



The **GUIDE Quick Start** dialog box contains two tab dialogs:

- **Create New GUI** – Asks you to start creating your new GUI by choosing a template for it. You can also specify the name by which the GUI is saved.

  See “Selecting a GUI Template” on page 6-5 for information about the templates.
- **Open Existing GUI** – Enables you to open an existing GUI in GUIDE. You can choose a GUI from your current directory or browse other directories.

Selecting a GUI Template

GUIDE provides several templates that you can modify to create your own GUIs. The templates are fully functional GUIs; they are already programmed.

This topic provides information for

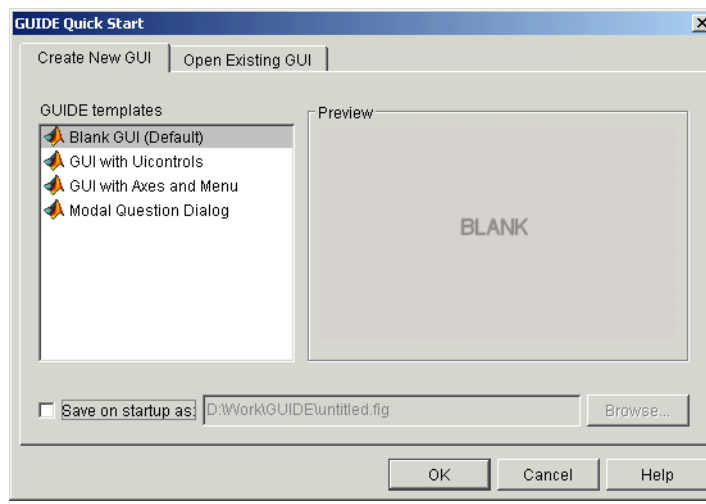
- "Accessing the Templates" on page 6-5
- "Template Descriptions" on page 6-6

Accessing the Templates

You can access the templates in two ways:

- Start GUIDE.
- If GUIDE is already open, select **New** from the **File** menu in the Layout Editor.

In either case, GUIDE displays the **GUIDE Quick Start** dialog box with the **Create New GUI** tab selected as shown in the following figure. This tab contains a list of the available templates.



To use a template:

1 Select a template in the left pane. A preview displays in the right pane.

2 Optionally, name your GUI now by selecting **Save on startup as** and typing the name in the field to the right. GUIDE saves the GUI before opening it in the Layout Editor. If you choose not to name the GUI at this point, GUIDE prompts you to save it and give it a name the first time you run the GUI.

3 Click **OK** to open the GUI template in the Layout Editor.

Template Descriptions

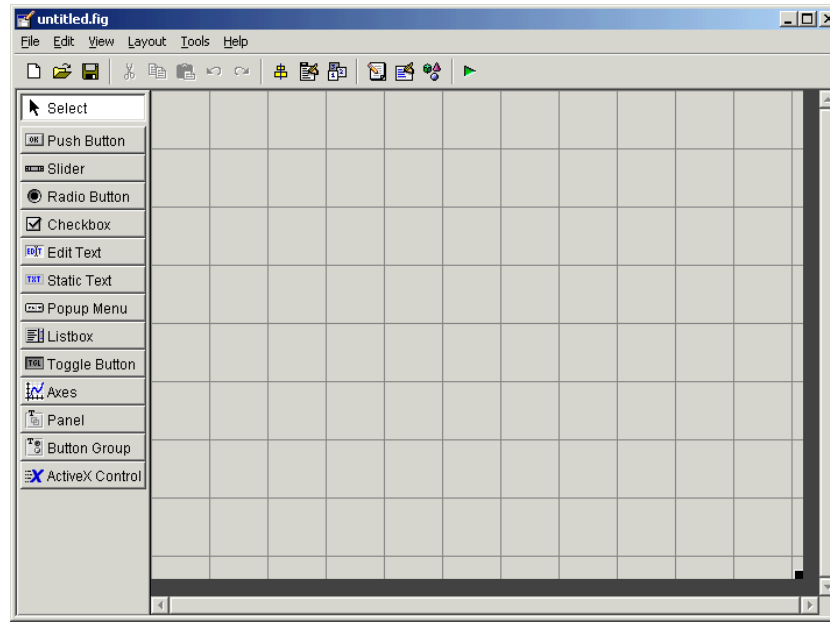
GUIDE provides four fully functional templates. They are described in the following sections:

- "Blank GUI" on page 6-6
- "GUI with Uicontrols" on page 6-7
- "GUI with Axes and Menu" on page 6-9
- "Modal Question Dialog" on page 6-10

Note To see how the template GUIs work, you can view their M-file code and look at their callbacks. You can also modify the callbacks for your own purposes. To view the M-file for any of these templates, open the template in the Layout Editor and click the M-file Editor icon on the toolbar. For information about using callbacks, see Chapter 8, "Programming the GUI."

Blank GUI

The blank GUI template displayed in the Layout Editor is shown in the following figure.

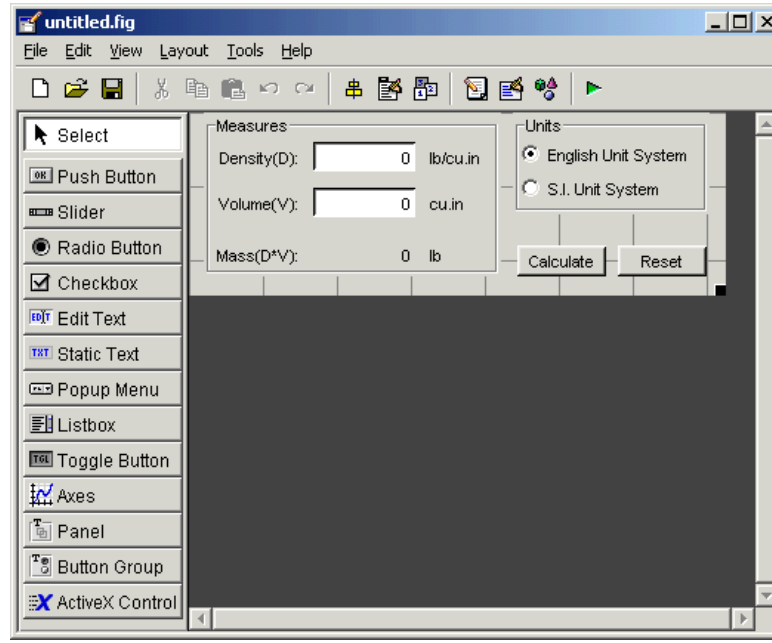


Select the blank GUI if the other templates are not suitable starting points for the GUI you are creating, or if you prefer to start with an empty GUI.

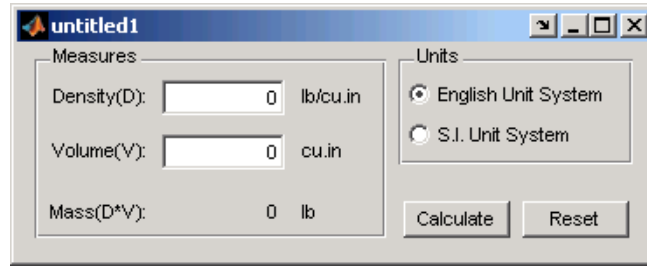
GUI with Uicontrols

The following figure shows the GUI with Uicontrols template displayed in the Layout Editor.

Note The term Uicontrols is short for user interface controls. These controls include push buttons, sliders, radio buttons, check boxes, editable and static text boxes, list boxes, and toggle buttons.



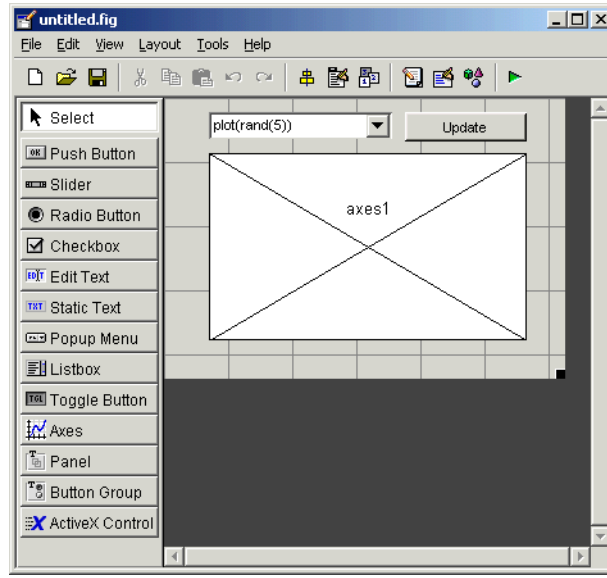
When you run the GUI by clicking the Run icon , the GUI appears as shown in the following figure.



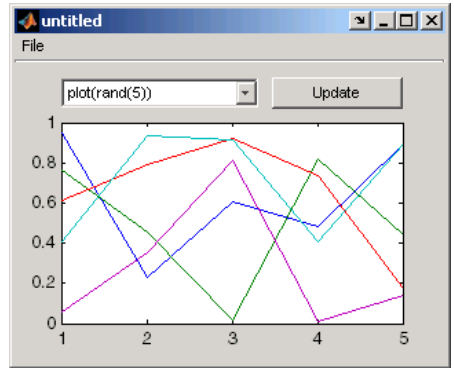
When a user enters values for the density and volume of an object, and clicks the **Calculate** button, the GUI calculates the mass of the object and displays the result next to `Mass(D*V)`.

GUI with Axes and Menu

The GUI with axes and menu template is shown in the following figure.



When you run the GUI by clicking the Run icon on the toolbar, the GUI displays a plot of five random numbers generated by the MATLAB `rand(5)` command, as shown in the following figure.



You can select other plots in the pop-up menu. Clicking the **Update** button displays the currently selected plot on the axes.

The GUI also has a **File** menu with three items:

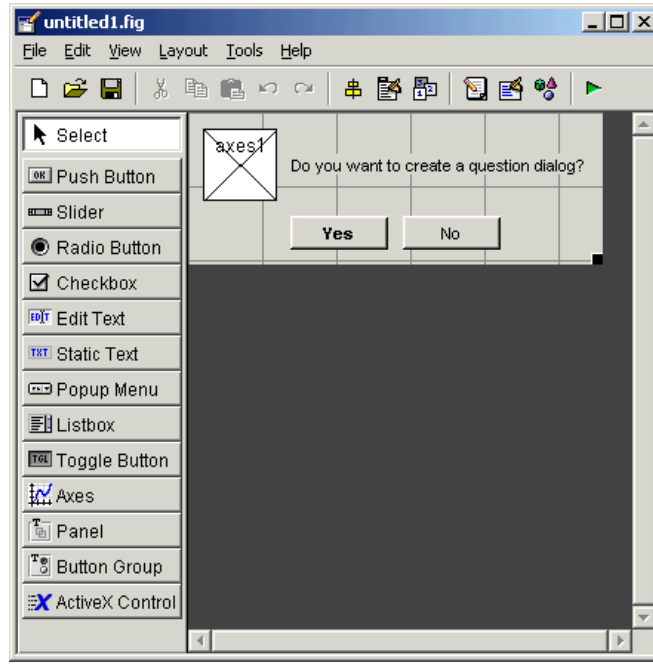
- Selecting **Open** displays a dialog from which you can open files on your computer.
- Selecting **Print** executes the `printdlg` command, which opens the **Print** dialog:

  ```
  printdlg(handles.figure1)
  ```

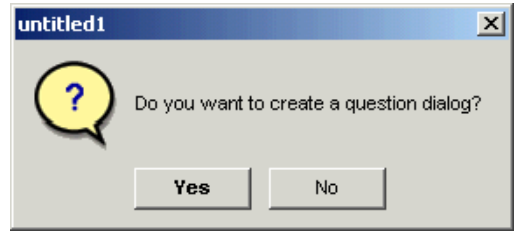
  Note that `handles.figure1` contains the current plot. Clicking **Yes** in the **Print** dialog prints the plot.
- Selecting **Close** closes the GUI.

Modal Question Dialog

The modal question dialog template displayed in the Layout Editor is shown in the following figure.



Running the GUI displays the dialog box shown in the following figure:



The GUI returns the text string `Yes` or `No`, depending on which button you press. The GUI is *blocking*, which means that the current M-file stops executing until the GUI restores execution. You can make a GUI blocking by adding the following command to the opening function:

```
uiwait(handles.figure1);
```

To restore access to other MATLAB windows once a button is clicked, add the following command to callbacks for both the **Yes** and **No** push buttons:

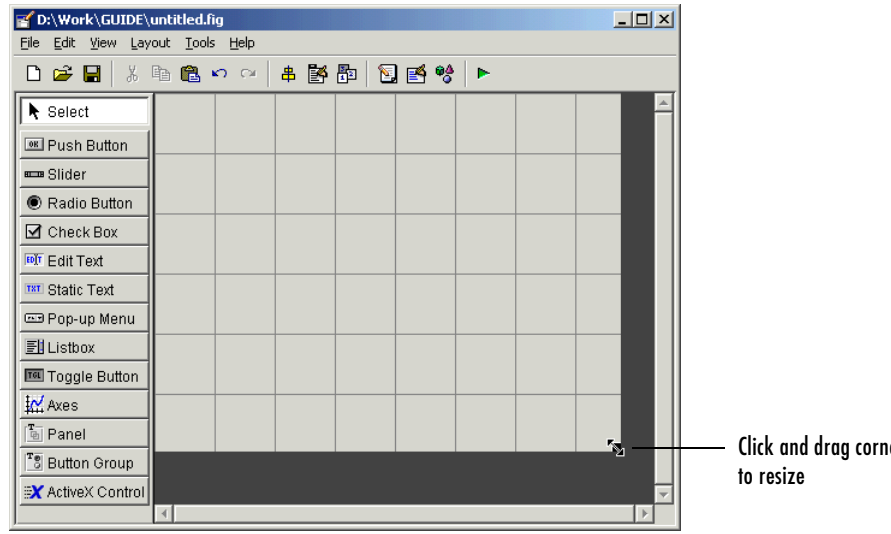
```
uiresume(handles.figure1);
```

The GUI is also *modal,* which means that the user cannot interact with other MATLAB windows until clicking one of the buttons. Select this template if you want your GUI to return a string or to be modal.

See “Using a Modal Dialog to Confirm an Operation” on page 10-47 for an example of using this template with another GUI. See the figure `WindowStyle` property for more information.

Setting the GUI Size

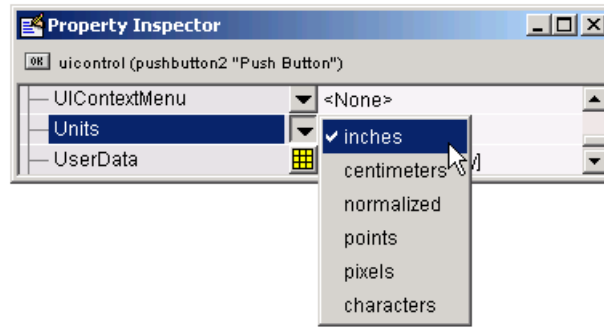
Set the size of the GUI by resizing the grid area in the Layout Editor. Click on the lower-right corner and drag it until the GUI is the desired size. If necessary, make the window larger.



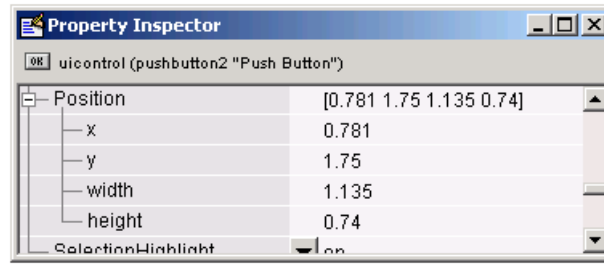
If you want to set the position or size of the GUI to an exact value, do the following:

1. Select **Property Inspector** from the **View** menu or by clicking the Property Inspector icon .

2. Scroll to the `Units` property and note whether the current setting is `characters` or `normalized`. Click the button next to `Units` and then change the setting to `inches` from the pop-up menu.



3 In the Property Inspector, click the **+** sign next to `Position`. The elements of the component's `Position` property are displayed.



4 Type the `x` and `y` coordinates of the point where you want the lower left corner of the GUI to appear, and its width and height.

5 Reset the `Units` property to its previous setting, either `characters` or `normalized`.

Note Setting the `Units` property to `characters` (nonresizable GUIs) or `normalized` (resizable GUIs) gives the GUI a more consistent appearance across platforms. See "Cross-Platform Compatible Units" on page 6-66 for more information.

Adding Components to the GUI

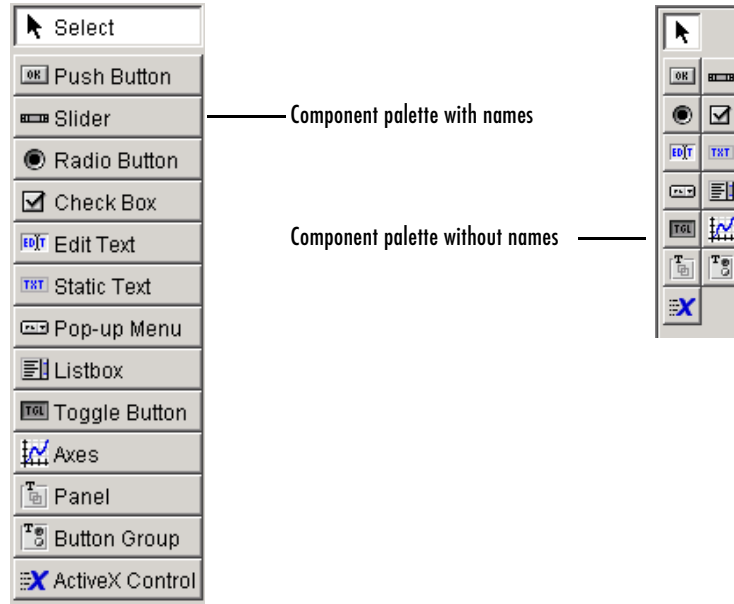
- "Available Components" on page 6-14
- "Adding Components to the Layout Area" on page 6-18
- "Assigning an Identifier to Each Component" on page 6-24
- "Adding Text to the Components" on page 6-25
- "Working with Components in the Layout Area" on page 6-31
- "Moving Components" on page 6-34
- "Resizing Components" on page 6-37

Other topics that may be of interest:

- "Aligning Components" on page 6-39
- "Setting Tab Order" on page 6-45

Available Components

The component palette, shown below, at the left side of the Layout Editor contains the components that you can add to your GUI. You can display it with or without names.



When you first open the Layout Editor, the component palette contains only icons. To display the names of the GUI components, select **Preferences** from the **File** menu, check the box next to **Show names in component palette**, and click **OK**.

Note This section provides information about using components to lay out a GUI. For information about programming these components see Chapter 8, "Programming the GUI."

Component	Description
Push Button	Push buttons generate an action when clicked. For example, an **OK** button might apply settings and close a dialog box. When you click a push button, it appears depressed; when you release the mouse, the button appears raised.
Toggle Button	Toggle buttons generate an action and indicate whether they are turned on or off. When you click a toggle button, it appears depressed, showing that it is on. When you release the mouse button. Unlike a push button, the toggle button remains depressed until you click the toggle button a second time. When you do so, the button returns to the raised state, showing that it is off.
Radio Button	Radio buttons are similar to check boxes, but are typically mutually exclusive within a group of related radio buttons. That is, you can select only one button at any given time. To activate a radio button, click the mouse button on the object. The display indicates the state of the button. Mutually exclusive radio buttons are managed using a button group.
Check Box	Check boxes generate an action when checked and indicate their state as checked or not checked. Check boxes are useful when providing the user with a number of independent choices that set a mode, for example, displaying a toolbar.
Edit Text	Edit text controls are fields that enable users to enter or modify text strings. Use edit text when you want text as input.
Static Text	Static text controls display lines of text. Static text is typically used to label other controls, provide directions to the user, or indicate values associated with a slider. Users cannot change static text interactively.

Component	Description
Slider	Sliders accept numeric input within a specific range by enabling the user to move a sliding bar, which is called a slider or thumb. Users move the slider by clicking the slider and dragging it, by clicking in the trough, or by clicking an arrow. The location of the slider indicates a percentage of the specified range.
List Box	List boxes display a list of items and enable users to select one or more items.
Pop-Up Menu	Pop-up menus open to display a list of choices when users click the arrow.
Axes	Axes enable your GUI to display graphics (e.g., graphs and images). Like all graphics objects, axes have properties that you can set to control many aspects of its behavior and appearance. See “Axes Properties” in the MATLAB documentation and commands such as the following for more information on axes objects: plot, surf, line, bar, polar, pie, contour and mesh. See Functions — By Category in the MATLAB documentation for a more complete list.
Panel	Panels group GUI components. By visually grouping related controls, panels can make the user interface easier to understand. A panel can have a title and various borders. Panel children can be user interface controls and axes as well as button groups and other panels. The position of each component within a panel is interpreted relative to the panel. If you move the panel, its children move with it and maintain their positions on the panel.

Component	Description
Button Group	Button groups are like panels but can be used to manage exclusive selection behavior for radio buttons and toggle buttons. For radio buttons and toggle buttons that are managed by a button group, you must include the code to control them in the button group's `SelectionChangeFcn` callback function, not in the individual user interface control (uicontrol) `Callback` functions. A button group overwrites the `Callback` properties of radio buttons and toggle buttons that it manages.
ActiveX Component	ActiveX components enable you to display ActiveX controls in your GUI. They are available only on the Microsoft Windows platform. An ActiveX control can be the child only of a figure, i.e., of the GUI itself. It cannot be the child of a panel or button group. See "ActiveX Control" on page 8-31in this document for an example. See MATLAB COM Client Support in the online MATLAB documentation to learn more about ActiveX controls.

Adding Components to the Layout Area

This topic tells you how to place components in the layout area and give each component a unique identifier.

1 Place a component in the layout area. Use either of these ways:

- Click the component in the component palette, then drag it into the layout area and drop it.
- Select the component in the component palette. The cursor changes to a cross. Add the component to the layout by either (1) placing the cursor in the layout area where you want the upper-left corner of the component to be and clicking, or (2) placing the cursor in the layout area where you want

the upper-left corner of the component to be, then setting the size of the control by dragging the cursor to the lower-right corner before releasing the mouse button.

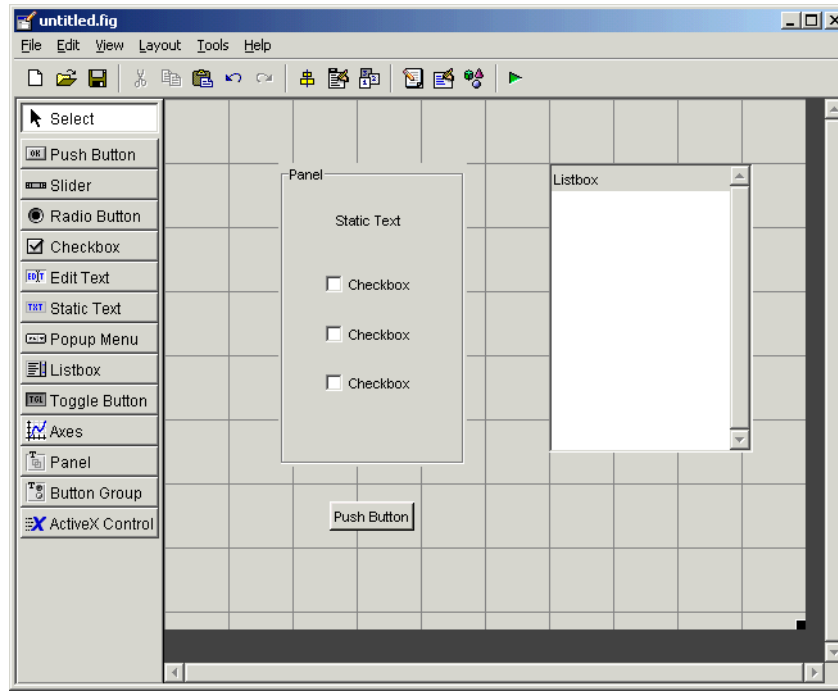
2. Assign a unique identifier to each component. Do this by setting the value of their `Tag` properties. See "Assigning an Identifier to Each Component" on page 6-24 for more information.

Note If you are adding an axes, allow space for tick marks to be placed outside the box that appears in the Layout Editor.

If you are adding a component to a panel or button group, you should also see "Adding a Component to a Panel or Button Group" on page 6-20.

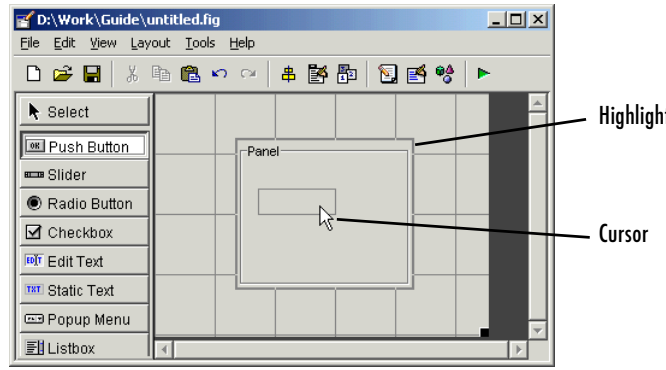
If you are adding an ActiveX control, you should also see "Adding an ActiveX Control" on page 6-22.

This is an example of a GUI in the Layout Editor. Note that components in the Layout Editor are not active. Chapter 7, "Saving and Running the GUI" describes how to generate a functioning GUI.

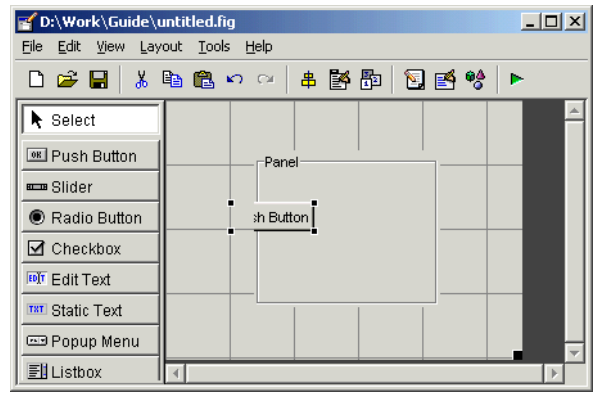


Adding a Component to a Panel or Button Group

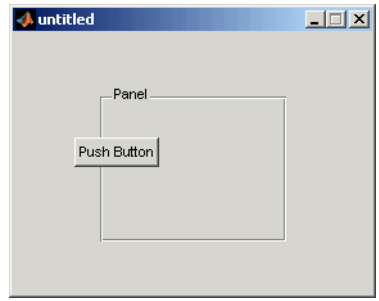
To add a component to a panel or button group, select the component in the component palette then move the cursor over the desired panel or button group. The position of the cursor determines the component's parent. Notice that GUIDE highlights the potential parent as shown in the following figure. The highlight indicates that if you drop the component or click the cursor, the component will be a child of the highlighted panel, button group, or figure.



If the component is not entirely contained in the panel or button group, it appears to be clipped in the layout editor.



When you run the GUI, the entire component is displayed and straddles the panel or button group border. The component is nevertheless a child of the panel and behaves accordingly.



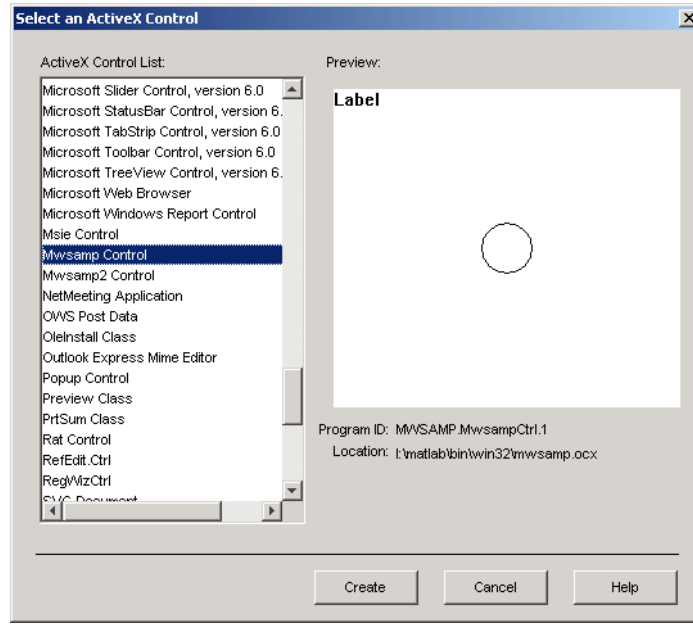
You can use the Object Browser to determine the child objects of a panel or button group. “Viewing the Object Hierarchy” on page 6-62 tells you how.

Note Assign a unique identifier to each component in your panel or button group by setting the value of their `Tag` properties. See “Assigning an Identifier to Each Component” on page 6-24 for more information.

Adding an ActiveX Control

When you drag an ActiveX component from the component palette into the layout area, GUIDE opens a dialog, similar to the following, that lists the registered ActiveX controls on your system.

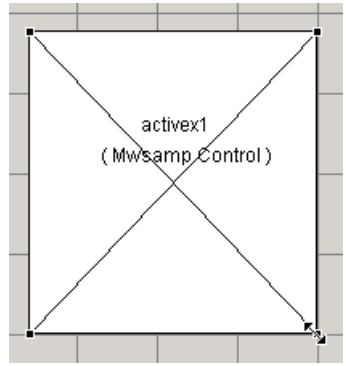
Note If MATLAB is not installed locally on your computer — for example, if you are running MATLAB over a network — you might not find the ActiveX control described in this example. To register the control, see Registering Controls and Servers in the online MATLAB documentation.



1 Select the desired ActiveX control. The right panel shows a preview of the selected control.

2 Click **Create**, the control appears as a small box in the Layout Editor.

3 Resize the control to approximately the size of the square shown in the preview pane. You can do this by clicking and dragging a corner of the control, as shown in the following figure.



See “ActiveX Control” on page 8-31 for information about programming a sample ActiveX control and an example.

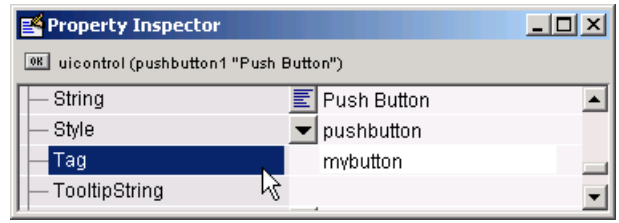
Assigning an Identifier to Each Component

Use the `Tag` property to assign each component a unique meaningful string identifier.

When you place a component in the layout area, GUIDE assigns a default value to the `Tag` property. Before saving the GUI, replace this value with a string that reflects the role of the component in the GUI.

The string value you assign `Tag` is used in the M-file code to identify the component and must be unique in the GUI. To set `Tag`:

1. Select **Property Inspector** from the **View** menu or by clicking the Property Inspector icon .
2. In the layout area select the component for which you want to set the `Tag`.
3. In the Property Inspector, select `Tag` and then replace the value with the string you want to use as the identifier. In the following figure, the `Tag` is set to `mybutton`.



Adding Text to the Components

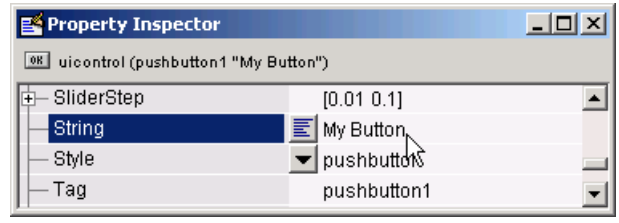
Many components have labels, lists, or other text that distinguishes them. You can set this text with the Property Inspector by setting the value of the appropriate property. In the Layout Area, select the component to which you want to add text. Then open the Property Inspector from the **View** menu or by clicking the Property Inspector icon . Subsequent topics tell you how to add text to specific components:

- "Push Button, Toggle Button, Radio Button, Check Box" on page 6-25
- "Pop-Up Menu" on page 6-26
- "Edit Text" on page 6-26
- "Static Text" on page 6-27
- "List Box" on page 6-28
- "Panel, Button Group" on page 6-30
- "Slider, Axes, ActiveX Control" on page 6-31

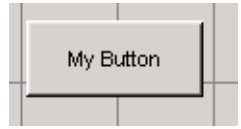
Push Button, Toggle Button, Radio Button, Check Box

These components have labels. To specify a label:

1 In the Property Inspector, select `String` and then replace the value with the text you want to use as the label.



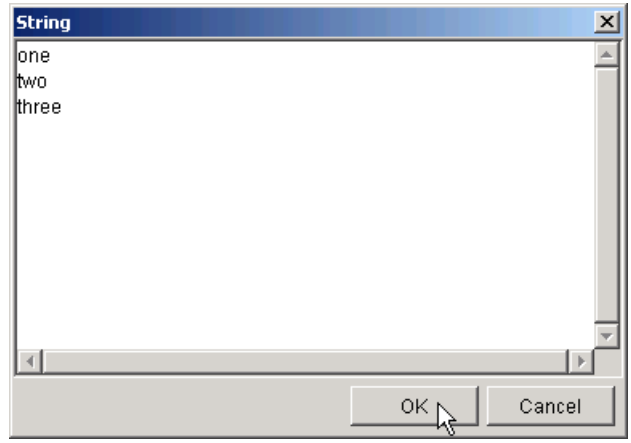
2 The component label changes when you click outside the `String` field.



Pop-Up Menu

Pop-up menus display a list of choices.

1 In the Property Inspector, click the button next to `String`. In the **String** dialog box that displays, replace the existing text with the list of choices in the order you want them to appear, then click **OK**.

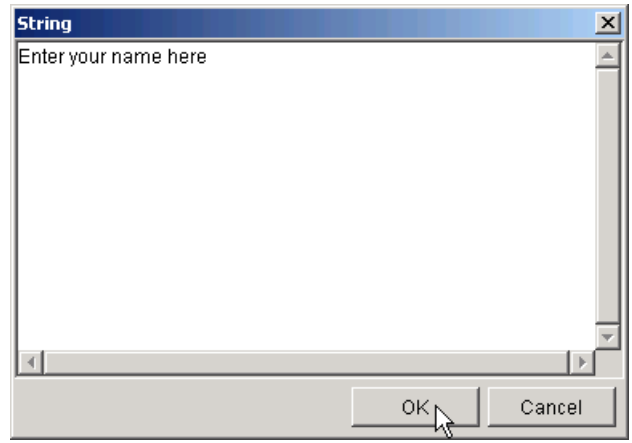


2 The first item in your list appears in the pop-up menu in the Layout Area.

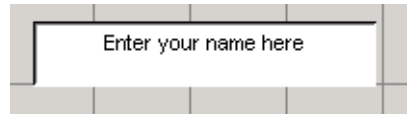
Edit Text

Edit text boxes enable users to enter or modify text strings, but you may want to set the initial text.

1 In the Property Inspector, click the button next to `String`. In the **String** dialog box that displays, replace the existing text with the text that you want to display when the edit text box first appears, then click **OK**. Use the **Enter** key to add additional lines after the first.



2 The text appears in the edit text box in the Layout Area.

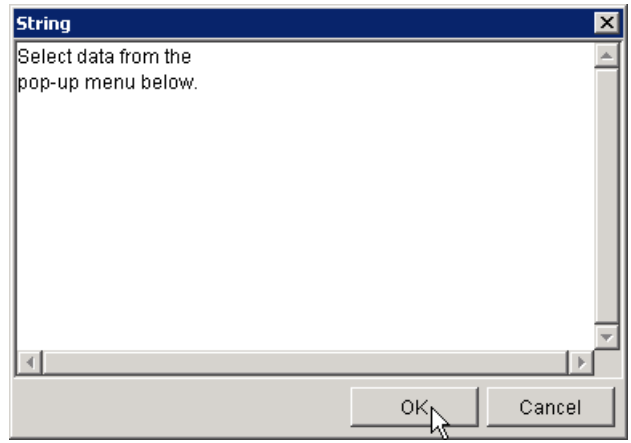


Note By default, the edit text box accepts only a single line of input. If you want it to accept multiple lines of input or if your initial text requires more than one line, use the Property Inspector to set the `Max` and `Min` properties so that `Max - Min > 1`. For example, `Max = 2`, `Min = 0`.

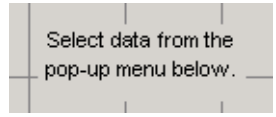
Static Text

Static text boxes display lines of text that the user cannot change.

1 In the Property Inspector, click the button next to `String`. In the **String** dialog box that displays, type the text that you want to display when that edit text box appears, then click **OK**. Use the **Enter** key to add additional lines after the first.



2 The text appears in the static text box in the Layout Area.

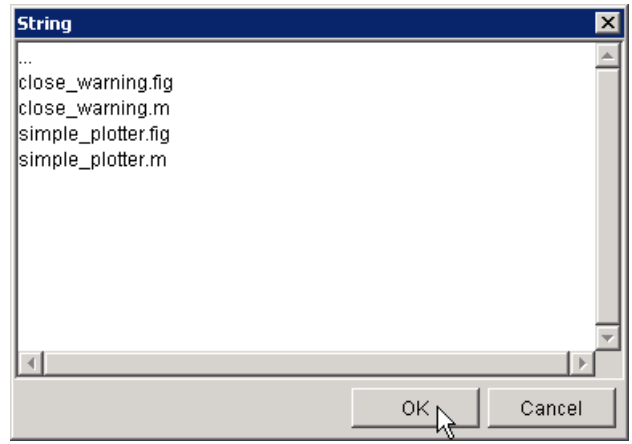


Note Unlike the edit text box and other user interface controls, the `Max` and `Min` properties do not affect the ability of a static text box to contain more than one line of text.

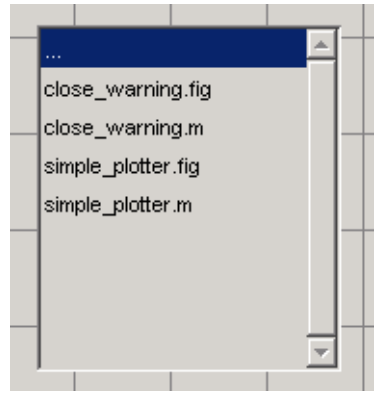
List Box

Displays a list from which a user can select one or more items.

1 In the Property Inspector, click the button next to `String`. In the **String** dialog box that displays, type the list of choices in the order you want them to appear, then click **OK**.



2 The list appears in the list box in the Layout Area with the top-most item selected.



3 If the list box is not large enough to display all list entries, you can specify which entry appears in the top-most position.

- Use the Property Inspector to set the `ListBoxTop` property to the index of the entry you want to appear at the top.
- By default, the list box enables a user to select only one item. If you want to allow multiple-item selection, use the Property Inspector to set the `Max`

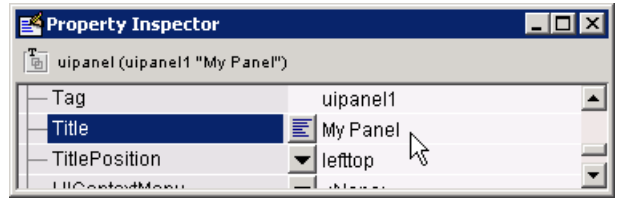
and `Min` properties so that `Max - Min > 1`. For example, `Max = 2`, `Min = 0`.

- Set the `Value` property to the index of the item you want selected when the list box is first displayed. If you want no initial selection, (1) set the `Value` property to an empty matrix by deleting the current value, and (2) set the `Max` and `Min` properties such that `Max - Min > 1`. This enables multiple selection, which is necessary if you want no initial selection.

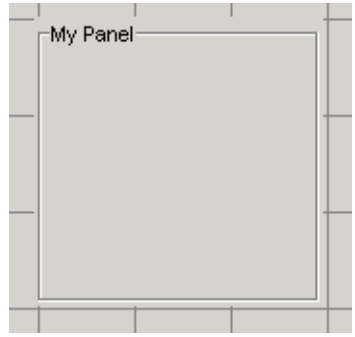
Panel, Button Group

The panel and button group enable you to group other components. You can specify a title for panels and button groups. You can also place the title at the top or bottom of the component, at the left, middle, or right.

1. In the Property Inspector, select `Title` and then replace the value with the text you want to use.



2. The panel or button group title changes when you click outside the `Title` field.



3 If you want to change the position of the title, select the button next to `TitlePosition` in the Property Inspector, and then select the desired position from the pop-up menu. The default is `lefttop`.

Slider, Axes, ActiveX Control

GUIDE provides no capability for associating text with sliders, axes, or with ActiveX controls.

- **Sliders** – Use a static text component to label a slider. Use an edit text box to enable a user to provide a value for the slider.
- **Axes** – Use the `title`, `xlabel`, `ylabel`, `zlabel`, and `text` functions to add text to an axes.
- **ActiveX controls** – For an ActiveX control, you can specify only those properties defined for it. To see the properties for an ActiveX control, click on the control in the Layout Area, then open the Property Inspector from the **View** menu or by clicking the Property Inspector icon .

Working with Components in the Layout Area

This topic provides basic information about selecting, copying, pasting, and deleting components in the layout area.

- "Selecting Components" on page 6-31
- "Copying, Cutting, and Clearing Components" on page 6-32
- "Pasting and Duplicating Components" on page 6-32
- "Front-to-Back Positioning" on page 6-33

Other topics that may be of interest are

- "Moving Components" on page 6-34
- "Resizing Components" on page 6-37
- "Aligning Components" on page 6-39
- "Setting Tab Order" on page 6-45

Selecting Components

You can select components in the layout area in the following ways.

- Click a single component to select it.

- Press **Ctrl+A** to select all child objects of the figure. This does not select components that are child objects of panels or button groups.
- Click and drag the cursor to create a rectangle that encloses the components you want to select. If the rectangle encloses a panel or button group, only the panel or button group is selected, not its children. If the rectangle encloses part of a panel or button group, only the components within the rectangle that are child objects of the panel or button group are selected.
- Select multiple components using the **Shift** and **Ctrl** keys.

In some cases, a component may lie outside its parent's boundary. Such a component is not visible in the Layout Editor but can be selected by dragging a rectangle that encloses it or by selecting it in the Object Browser. Such a component is visible in the active GUI.

Note You can select multiple components only if they have the same parent. To determine the child objects of a figure, panel, or button group, use the Object Browser. "Viewing the Object Hierarchy" on page 6-62 tells you how

---

Copying, Cutting, and Clearing Components

Use standard menu and pop-up menu commands, toolbar icons, keyboard keys, and shortcut keys to copy, cut, and clear components.

Copying. Copying places a copy of the selected components on the clipboard. A copy of a panel or button group includes its children.

Cutting. Cutting places a copy of the selected components on the clipboard and deletes them from the layout area. If you cut a panel or button group, you also cut its children.

Clearing. Clearing deletes the selected components from the layout area. It does not place a copy of the components on the clipboard. If you clear a panel or button group, you also clear its children.

Pasting and Duplicating Components

Pasting. Use standard menu and pop-up menu commands, toolbar icons, and short-cut keys to paste components. GUIDE pastes the contents of the

clipboard to the location of the last mouse click. It positions the upper left corner of the contents at the mouse click

Consecutive pastes place each copy to the lower right of the last one.

Duplicating. Select one or more components that you want to duplicate, then do one of the following:

- Copy and paste the selected components as described above.
- Select **Duplicate** from the **Edit** menu or the pop-up menu. **Duplicate** places the copy to the lower right of the original.
- Right-click and drag the component to the desired location. The position of the cursor when you drop the components determines the parent of all the selected components. Look for the highlight as described in "Adding a Component to a Panel or Button Group" on page 6-20.

Front-to-Back Positioning

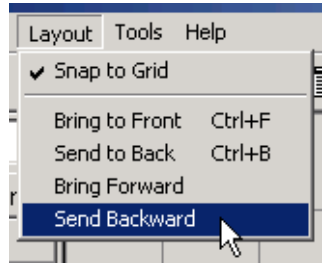
MATLAB figures maintain separate stacks that control the front-to-back positioning for different kinds of components:

- User interface controls such as buttons, sliders, and pop-up menus
- Panels, button groups, and axes
- ActiveX controls

You can control the front-to-back positioning of components that overlap only if those components are in the same stack. For overlapping components that are in different stacks:

- User interface controls always appear on top of panels, button groups, axes, and ActiveX controls they overlap.
- Panels, button groups, and axes always appear on top of ActiveX controls.

The Layout Editor provides four operations that enable you to control front-to-back positioning. All are available from the **Layout** menu, which is shown in the following figure.



- **Bring to Front** — move the selected object(s) in front of nonselected objects (available from the right-click context menu, the **Layout** menu, or the **Ctrl+F** shortcut).
- **Send to Back** — move the selected object(s) behind nonselected objects (available from the right-click context menu, the **Layout** menu, or the **Ctrl+B** shortcut).
- **Bring Forward** — move the selected object(s) forward by one level, i.e., in front of the object directly forward of it, but not in front of all objects that overlay it (available from the **Layout** menu).
- **Send Backward** — move the selected object(s) back by one level, i.e., behind the object directly in back of it, but not behind all objects that are behind it (available from the **Layout** menu).

Note Changing front-to-back positioning of components also changes their tab order. See "Setting Tab Order" on page 6-45 for more information.

Moving Components

You can move components in one of the following ways:

- "Dragging Components" on page 6-35
- "Using Arrow Keys to Move Components" on page 6-35
- "Setting the Component's Position Property" on page 6-35

Other topics that may be of interest are

- "Aligning Components" on page 6-39

Dragging Components

Select one or more components that you want to move, then drag them to the desired position and drop them. You can move components from the figure into a panel or button group. You can move components from a panel or button group into the figure or into another panel or button group.

The position of the cursor when you drop the components determines the parent of all the selected components. Look for the highlight as described in "Adding a Component to a Panel or Button Group" on page 6-20.

In some cases, one or more of the selected components may lie outside its parent's boundary. Such a component is not visible in the Layout Editor but can be selected by dragging a rectangle that encloses it. It is visible, however, in the active GUI.

Note To select multiple components, they must have the same parent. That is, they must be contained in the same figure, panel, or button group. See "Selecting Components" on page 6-31 for more information.

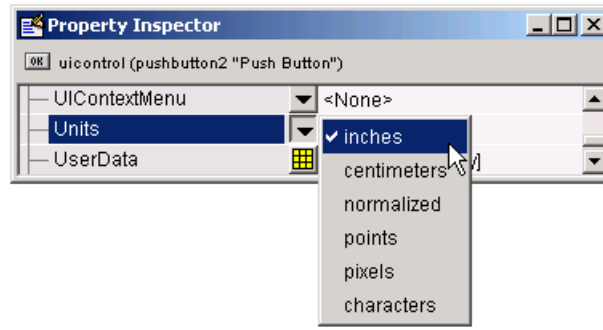
Using Arrow Keys to Move Components

Select one or more components that you want to move, then press and hold the arrow keys until the components have moved to the desired position. Note that the components remain children of the figure, panel, or button group from which you move them, even if they move outside its boundaries.

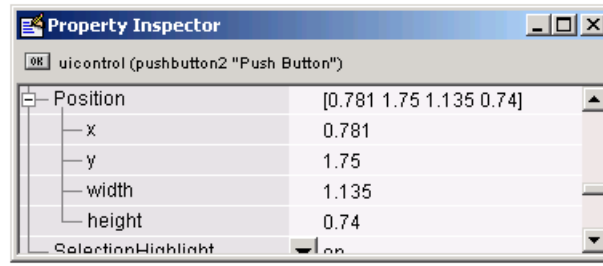
Setting the Component's Position Property

Select one or more components that you want to move. Then open the Property Inspector from the **View** menu or by clicking the Property Inspector icon .

1. In the Property Inspector, scroll to the `Units` property and note whether the current setting is `characters` or `normalized`. Click the button next to `Units` and then change the setting to `inches` from the pop-up menu.



2 Click the + sign next to `Position`. The Property Inspector displays the elements of the `Position` property.



3 If you have selected only one component, type the `x` and `y` coordinates of the point where you want the lower left corner of the component to appear.

If you have selected more than one component, type either the `x` or the `y` coordinate to align the components along that dimension.

4 Reset the `Units` property to its previous setting, either `characters` or `normalized`.

Note Setting the `Units` property to `characters` (nonresizable GUIs) or `normalized` (resizable GUIs) gives the GUI a more consistent appearance across platforms. See "Cross-Platform Compatible Units" on page 6-66 for more information.

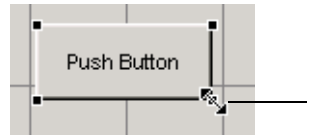
Resizing Components

You can resize components in one of the following ways:

- "Dragging a Corner of the Component" on page 6-37
- "Setting the Component's Position Property" on page 6-37

Dragging a Corner of the Component

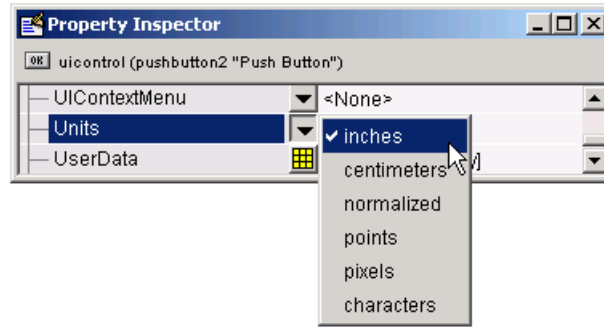
Select the component you want to resize. Click on the one of the corner handles and drag it until the component is the desired size.



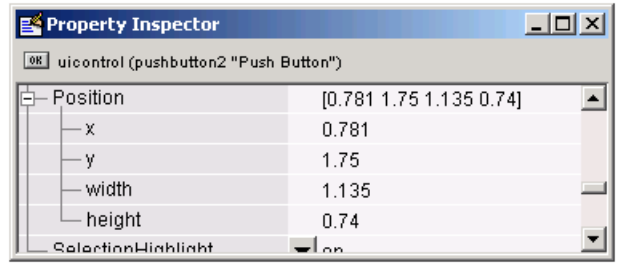
Setting the Component's Position Property

Select one or more components that you want to resize. Then open the Property Inspector from the **View** menu or by clicking the Property Inspector icon .

1. In the Property Inspector, scroll to the `Units` property and note whether the current setting is `characters` or `normalized`. Click the button next to `Units` and then change the setting to `inches` from the pop-up menu.



2. Click the **+** sign next to `Position`. The Property Inspector displays the elements of the `Position` property.



3 Type the `width` and `height` you want the components to be.

4 Reset the `Units` property to `characters`.

5 Reset the `Units` property to its previous setting, either `characters` or `normalized`.

Note To select multiple components, they must have the same parent. That is, they must be contained in the same figure, panel, or button group. See "Selecting Components" on page 6-31 for more information.

Note Setting the `Units` property to `characters` (nonresizable GUIs) or `normalized` (resizable GUIs) gives the GUI a more consistent appearance across platforms. See "Cross-Platform Compatible Units" on page 6-66 for more information.

Aligning Components

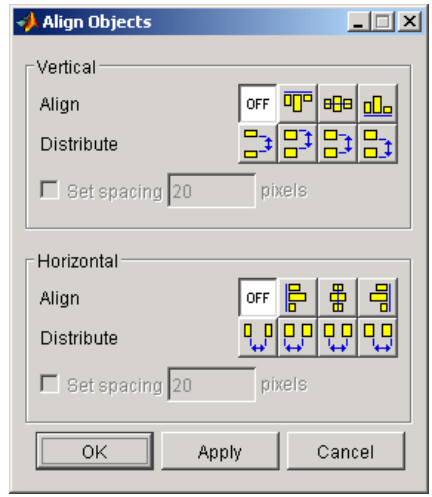
The Layout Editor provides a number of features that facilitate alignment and distribution of components with respect to one another:

- "Alignment Tool" on page 6-39 — align and distribute groups of components.
- "Property Inspector" on page 6-41 — align components by modifying their `Position` properties.
- "Grid and Rulers" on page 6-42 — align components on a grid with optional snap to grid.
- "Guide Lines" on page 6-43 — vertical and horizontal snap-to guides at arbitrary locations.

Alignment Tool

The Alignment Tool enables you to position objects with respect to each other and to adjust the spacing between selected objects. The specified alignment operations apply to all components that are selected when you press the **Apply** button.

Note To select multiple components, they must have the same parent. That is, they must be contained in the same figure, panel, or button group. See "Selecting Components" on page 6-31 for more information.



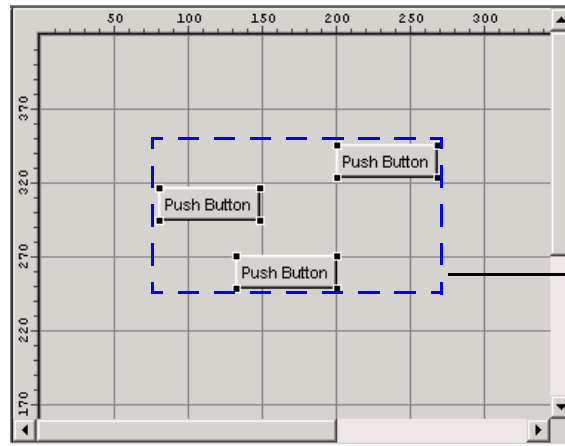
The alignment tool provides two types of alignment operations:

- **Align** — align all selected components to a single reference line.
- **Distribute** — space all selected components uniformly with respect to each other.

Both types of alignment can be applied in the vertical and horizontal directions. Note that, in many cases, it is better to apply alignments independently to the vertical or to the horizontal using two separate steps.

Align Options

There are both vertical and horizontal align options. Each option aligns selected components to a reference line, which is determined by the bounding box that encloses the selected objects. For example, the following picture of the layout area shows the bounding box (indicated by the dashed line) formed by three selected push buttons.



All of the align options (vertical top, center, bottom and horizontal left, center, right) place the selected components with respect to the corresponding edge (or center) of this bounding box.

Distribute Options

Distributing components adds equal space between all components in the selected group. The distribute options operate in two different modes:

- Equally space selected components within the bounding box (default)
- Space selected components to a specified value in pixels (check **Set spacing** and specify a pixel value)

Both modes enable you to specify how the spacing is measured, as indicated by the button labels on the alignment tool. These options include spacing measured with respect to the following edges:

- Vertical — inner, top, center, and bottom
- Horizontal — inner, left, center, and right

Property Inspector

The Property Inspector enables you to align components by setting their `Position` properties. A component's `Position` property is a 4-element vector that specifies the location of the component on the GUI and its size: [distance

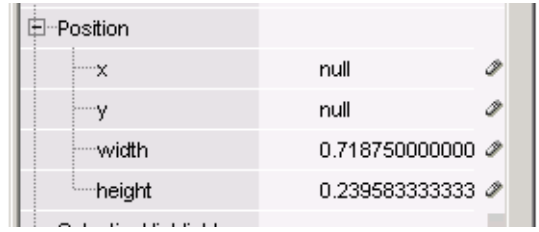
from left, distance from bottom, height, width]. The values are given in the units specified by the `Units` property of the component.

1. Select the components you want to align. See "Selecting Components" on page 6-31 for information.

2. Select **Property Inspector** from the **View** menu or by clicking the Property Inspector icon .

3. In the Property Inspector, scroll to the `Units` property and note its current setting, then change the setting to `inches`.

4. Scroll to the `Position` property. A `null` value means that the element differs in value for the different components. This figure shows the `Position` property for multiple components of the same size.

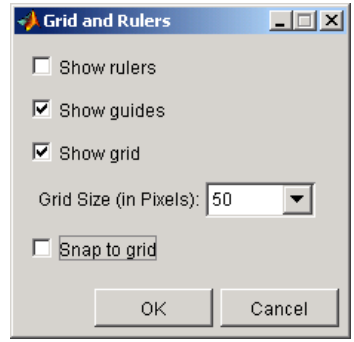


5. Change the value of `x` to align their left sides. Change the value of `y` to align their bottom edges. For example, setting `x` to 2.0 aligns the left sides of the components 2 inches from the left side of the GUI.

6. When the components are aligned, change the `Units` property back to its original setting.

Grid and Rulers

The layout area displays a grid and rulers to facilitate component layout. Grid lines are spaced at 50-pixel intervals by default and you can select from a number of other values ranging from 10 to 200 pixels. You can optionally enable *snap-to-grid*, which causes any object that is moved or resized to within 9 pixels of a grid line to jump to that line. Snap-to-grid works with or without a visible grid.



Use the Grid and Rulers dialog (select **Grid and Rulers** from the **Tools** menu) to:

- Control visibility of rulers, grid, and guide lines
- Set the grid spacing
- Enable or disable snap-to-grid

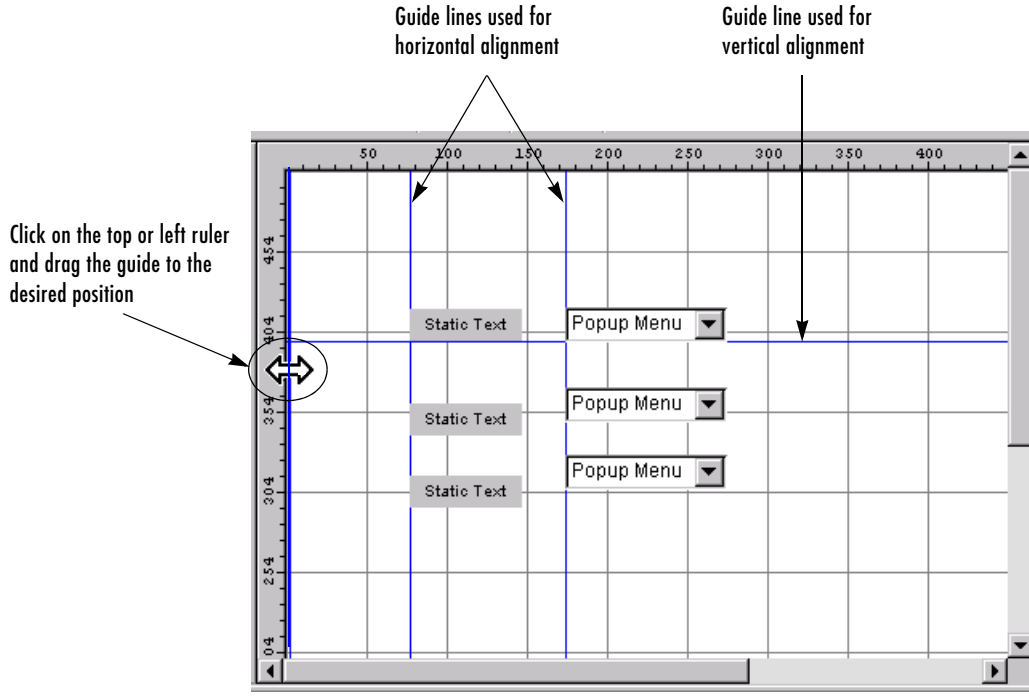
Guide Lines

The Layout Editor has both vertical and horizontal snap-to guide lines. Components snap to the line when you move or resize them to within nine pixels of the line.

Guide lines are useful when you want to establish a reference for component alignment at an arbitrary location in the Layout Editor.

Creating Guide Lines

To create a guide line, click on the top or left ruler and drag the line into the layout area.


Guide lines used for horizontal alignment
Guide line used for vertical alignment
Click on the top or left ruler and drag the guide to the desired position
Static Text
Popup Menu
Static Text
Popup Menu
Static Text
Popup Menu

Setting Tab Order

A GUI's tab order is the order in which components of the GUI are selected when a user presses the **Tab** key on the keyboard.

MATLAB figures maintain separate stacks that control tabbing for different kinds of components:

- User interface controls such as buttons, sliders, and pop-up menus
- Panels, button groups (axes are in the same stack but cannot be tabbed)
- ActiveX controls (cannot be tabbed)

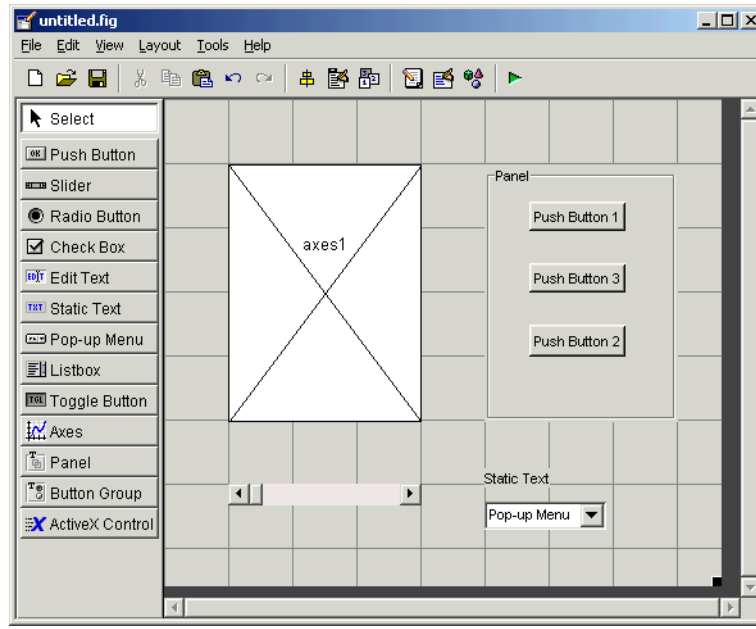
You can control the tab order of components only if those components are in the same stack.

The tab order for each level is determined independently. The figure is the base level, and each panel or button group establishes its own level. If, in tabbing through the components at the figure level, a user tabs to a panel or button group, then subsequent tabs sequence through the components of the panel or button group before returning to the level from which the panel or button group was reached.

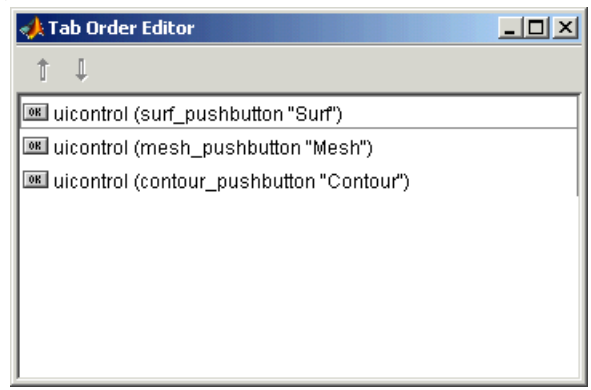
When you create a GUI, GUIDE sets the tab order at each level to be the order in which you add components to that level in the Layout Editor. This may not be the best order for the user.

Note The tab order also affects the stacking order of components. If components overlap, those that appear lower in the tabbing order, are drawn on top of those that appear higher in the order. See "Front-to-Back Positioning" on page 6-33 for more information.

The figure in the following GUI contains an axes, a slider, a panel, a static text, and a pop-up menu. Of these, only the slider, the panel, and the pop-up menu at the figure level can be tabbed. The panel contains three push buttons, which can all be tabbed.



To examine and change the tab order of the panel components, click the panel background to select it, then select **Tab Order Editor** in the **Tools** menu of the Layout Editor.



The Tab Order Editor displays the panel's components in their current tab order. To change the tab order, select a component and press the **up** or **down**

arrow to move the component up or down in the list. If you set the tab order for the three components in the example to be

1. **Surf** push button
2. **Contour** push button
3. **Mesh** push button

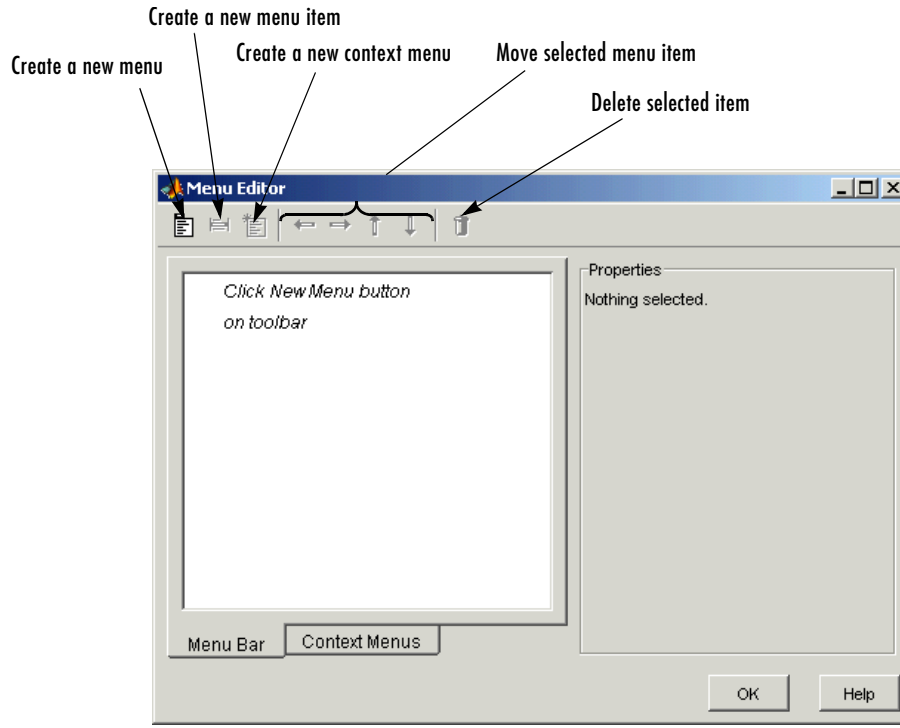
the user first tabs to the **Surf** push button, then to the **Contour** push button and then to the **Mesh** push button. Subsequent tabs sequence through the remaining components at the figure level.

Creating Menus

GUIDE enables you to create two kinds of menus:

- "Menus for the Menu Bar" on page 6-49 — drop-down menus whose titles appear on the figure menu bar.
- "Context Menus" on page 6-56 — pop up when users right-click on graphics objects. They are also known as shortcut menus.

You can create both types of menus using the Menu Editor. Access the Menu Editor from the **Tools** menu or from the Layout Editor toolbar.



Note In general, programming conventions described for components in Chapter 8, “Programming the GUI” also apply to menu items. See “Menu Item” on page 8-37 and “Updating a Menu Item Check” on page 8-38 for information about programming and basic examples.

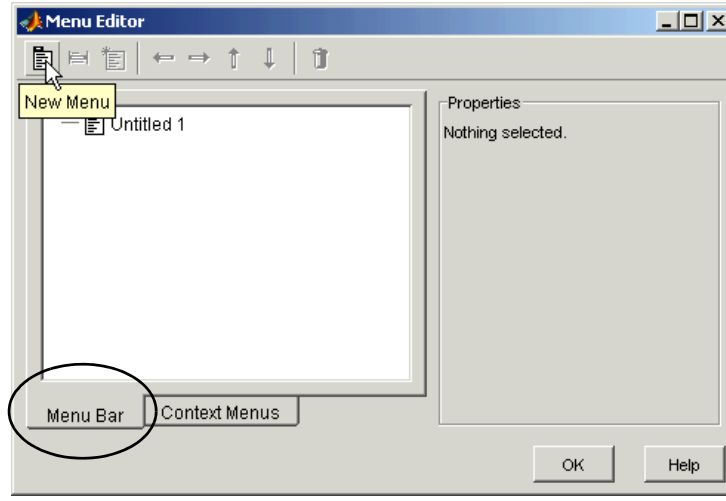
Menus for the Menu Bar

When you create a drop-down menu, GUIDE adds its title to the GUI menu bar. You can then create menu items for that menu. Each menu item can have a cascading menu, also known as a submenu, and these items can have cascading menus, and so on.

When you create the first drop-down menu, GUIDE automatically adds a menu bar to the GUI.

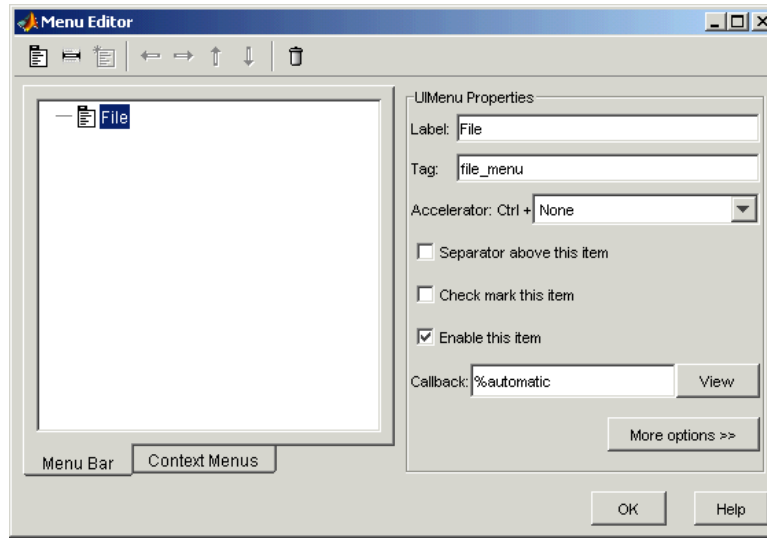
Creating a Menu

1. Start a new menu by clicking the **New Menu** tool. A menu title, `Untitled 1`, appears in the left pane of the dialog box.



Note By default, GUIDE selects the **Menu Bar** tab when you open the Menu Editor.

2 Click on the menu title to display a selection of menu properties in the right pane.



3 Fill in the **Label** and **Tag** fields for the menu. For example, set **Label** to `'File'` and set **Tag** to `'file_menu'`. Click outside the field for the change to take effect.

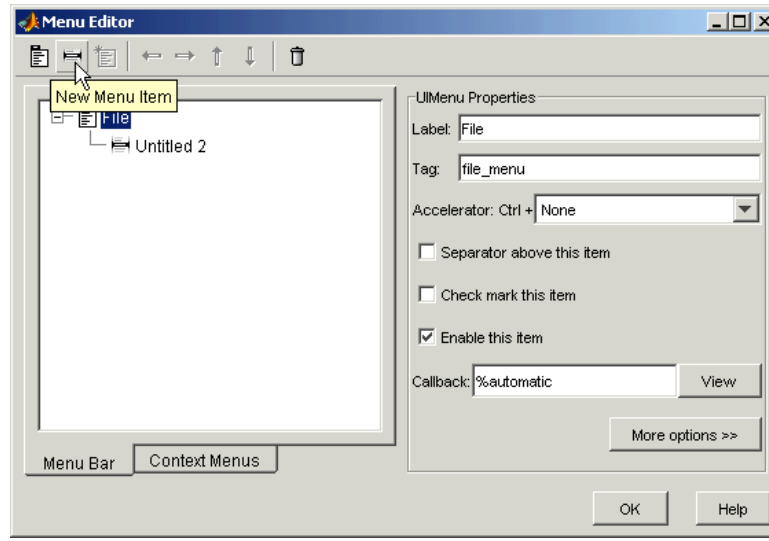
Label is a string that specifies the text label for the menu item. To display the "&" character in a label, use two `'&&'` characters in the string. The words `remove` and `default` (case-sensitive) are reserved. To use either of these as labels, prepend a backslash (`\`) to the string. For example, `'\remove'` yields **remove.**

Tag is a string that is an identifier for the menu object. It is used in the code to identify the menu item and must be unique in the GUI.

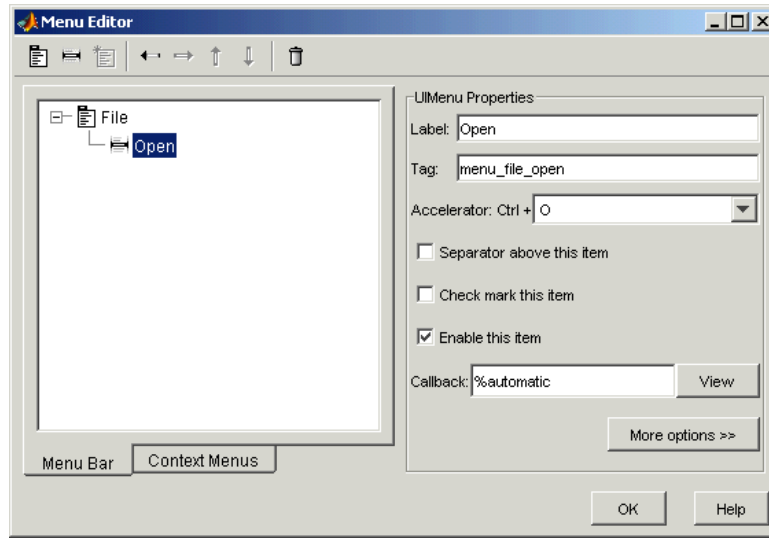
Adding Items to a Menu

Use the **New Menu Item** tool to create menu items that are displayed in the drop-down menu.

1. Add an **Open** menu item under `File`, by selecting `File` then clicking the **New Menu Item** tool. A temporary numbered menu item label, `'Untitled'`, appears.



2. Fill in the **Label** and **Tag** fields for the new menu item. For example, set **Label** to `'Open'` and set **Tag** to `'menu_file_open'`. Click outside the field for the change to take effect.



You can also:

- Choose a keyboard accelerator for the menu item with the **Accelerator** pop-up menu. In combination with **Ctrl**, this is the keyboard equivalent for a menu item that does not have a child menu. Note that some accelerators may be used for other purposes on your system and that other actions may result.
- Display a separator above the menu item by checking **Separator above this item**.
- Display a check next to the menu item when the menu is first opened by checking **Check mark this item**. A check indicates the current state of the menu item. See the example in “Adding Items to the Context Menu” on page 6-57.
- Enable this item when the menu is first opened by checking **Enable this item**. This allows the user to select this item when the menu is first opened. If you deselect this option, the menu item appears dimmed when the menu is first opened, and the user cannot select it.
- Specify a string for the routine, i.e., the **Callback**, that performs the action associated with the menu item. If you have not yet saved the GUI, the default value is `'%automatic'`. When you save the GUI, and if you have not changed

this field, GUIDE automatically sets the value using a combination of the **Tag** field and the GUI filename. See “Menu Item” on page 8-37 for more information about specifying this field and for programming menu items.

The **View** button displays the callback, if there is one, in an editor. If you have not yet saved the GUI, GUIDE prompts you to save it.

- Open the Property inspector, where you can change all menu properties, by clicking the **More options** button.

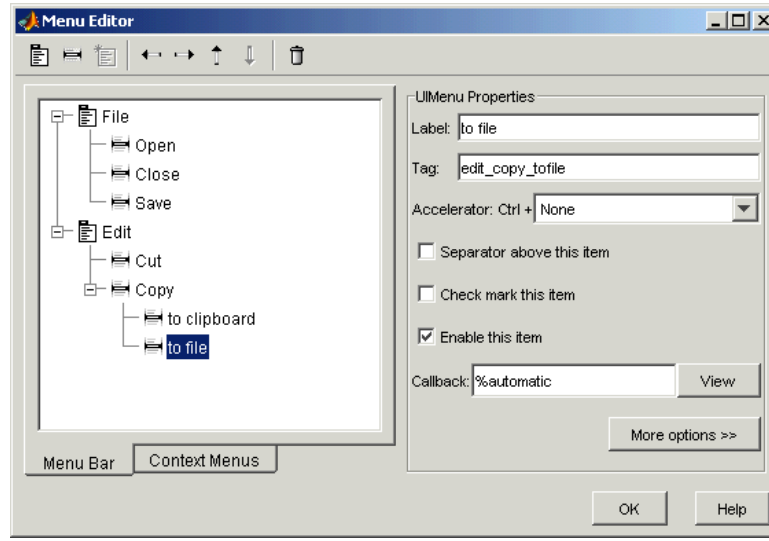
Note In general, programming conventions described for components in Chapter 8, “Programming the GUI” also apply to menu items. See “Menu Item” on page 8-37 and “Updating a Menu Item Check” on page 8-38 for programming information and basic examples.

Additional Drop-Down Menus

To create additional drop-down menus, use the **New Menu** tool in the same way you did to create the `File` menu. For example, the following figure also shows an `Edit` drop-down menu.

Cascading Menus

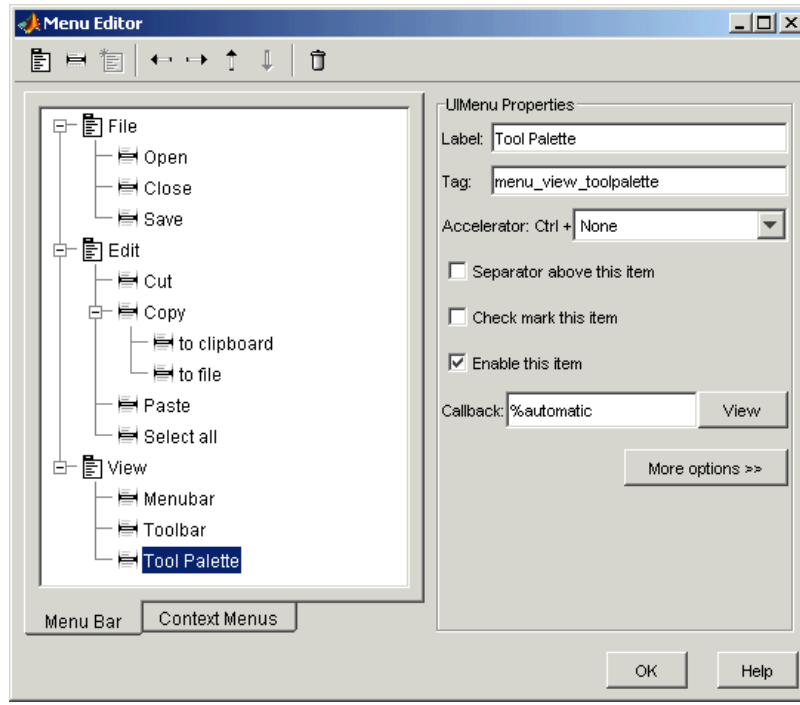
To create a cascading menu, select the menu item that will be the title for the cascading menu, then click the **New Menu Item** tool. In the example below, `Copy` is a cascading menu.



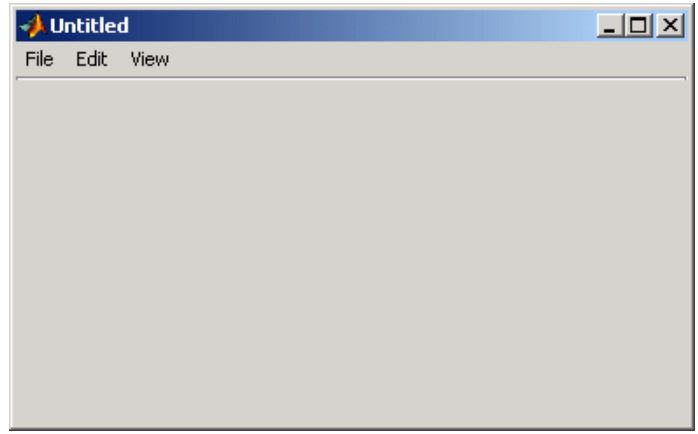
Note See "Menu Item" on page 8-37 for information about programming menu items.

Laying Out Three Menus

The following Menu Editor illustration shows three menus defined for the figure menu bar.



When you run the GUI, the menu titles appear in the menu bar.



Context Menus

A context menu is displayed when a user right-clicks on the object for which the menu is defined. The Menu Editor enables you to define context menus and associate them with objects in the layout.

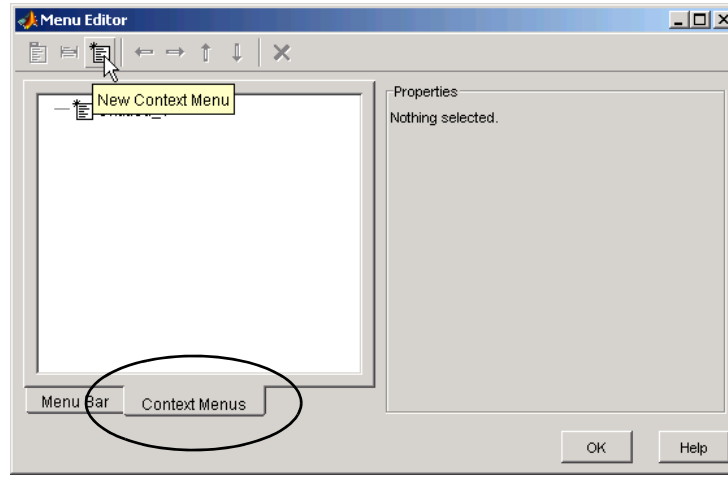
See “Menus for the Menu Bar” on page 6-49 for information about defining menus in general.

Note See “Menu Item” on page 8-37 for information about defining callback subfunctions for your menus.

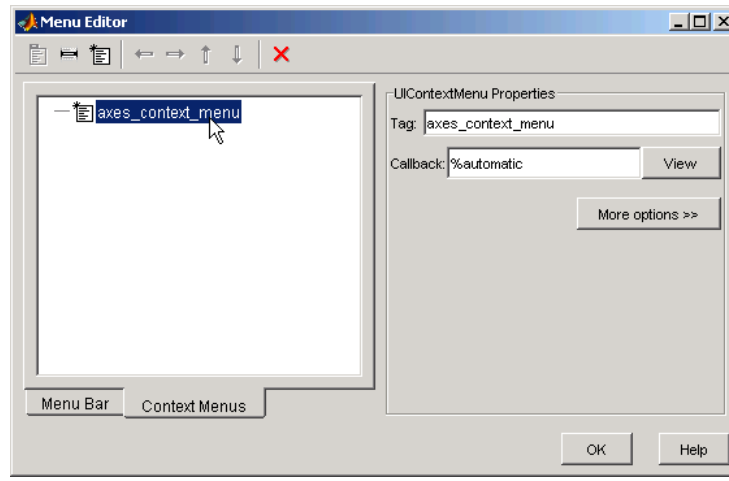
Creating the Parent Menu

All items in a context menu are children of a menu that is not displayed on the figure menu bar. To define the parent menu

1. Select the Menu Editor’s **Context Menus** tab and select **New Context Menu** from the toolbar.



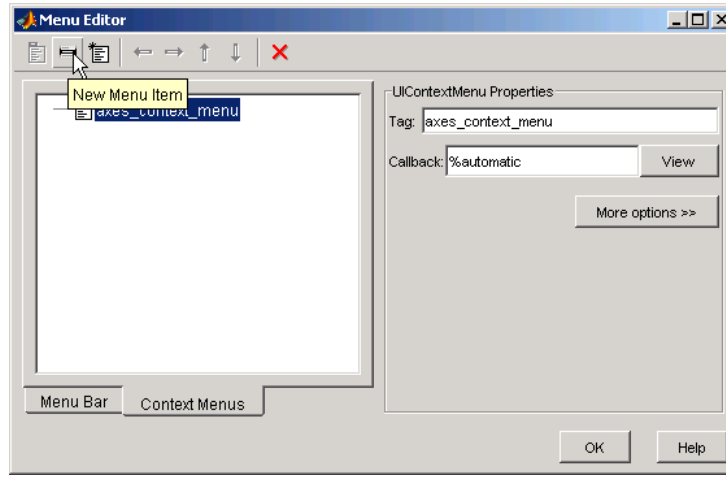
2. Select the menu and specify the `Tag` to identify the context menu (`axes_context_menu` in this example).



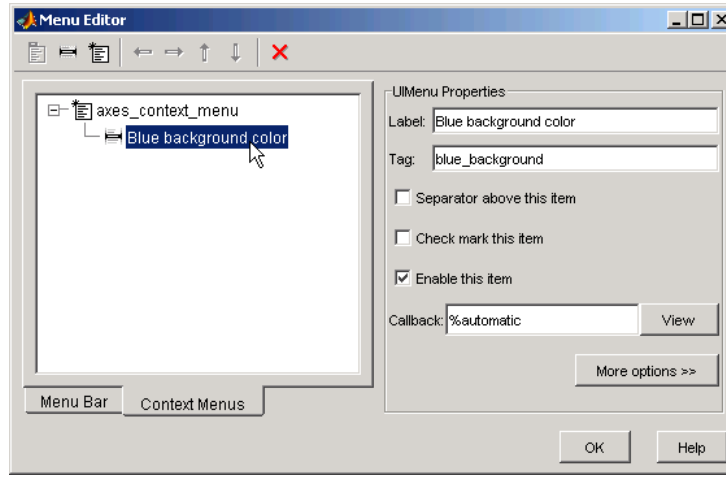
Adding Items to the Context Menu

Use the **New Menu Item** tool to create menu items that are displayed in the context menu.

1. Add a **Blue background color** menu item to the menu by selecting `axes_context_menu` and clicking the **New Menu Item** tool. A temporary numbered menu item label, `'Untitled'`, appears.



2 Fill in the **Label** and **Tag** fields for the new menu item. For example, set **Label** to `'Blue background color'` and set **Tag** to `'blue_background'`. Click outside the field for the change to take effect.



You can also:

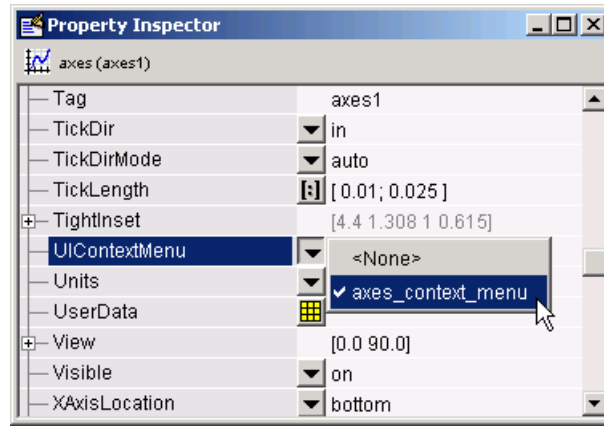
- Display a separator above the menu item by checking **Separator above this item**.

- Display a check next to the menu item when the menu is first opened by checking **Check mark this item**. A check indicates the current state of the menu item. See the example in "Adding Items to the Context Menu" on page 6-57. See "Updating a Menu Item Check" on page 8-38 for an example.
- Enable this item when the menu is first opened by checking **Enable this item**. This allows the user to select this item when the menu is first opened. If you deselect this option, the menu item appears dimmed when the menu is first opened, and the user cannot select it.
- Specify a string for the routine, i.e., the **Callback**, that performs the action associated with the menu item. If you have not yet saved the GUI, the default value is `'%automatic'`. When you save the GUI, and if you have not changed this field, GUIDE automatically sets the value using a combination of the **Tag** field and the GUI filename. See "Menu Item" on page 8-37 for more information about specifying this field and for programming menu items.

  The **View** button displays the callback, if there is one, in an editor. If you have not yet saved the GUI, GUIDE prompts you to save it.
- Open the Property inspector, where you can change all menu properties, by clicking the **More options** button.

Associating the Context Menu with an Object

In the Layout Editor, select the object for which you are defining the context menu. Use the Property Inspector to set this object's `UIContextMenu` property to the name of the desired context menu. The following figure shows the `UIContextMenu` property for the `axes` object with `Tag` property `axes1`.



In the GUI M-file, complete the callback subfunction for each item in the context menu. Each callback executes when a user selects the associated context menu item. See “Menu Item” on page 8-37 for information on defining the syntax.

Note In general, programming conventions described for components in Chapter 8, “Programming the GUI” also apply to menu items. See “Menu Item” on page 8-37 and “Updating a Menu Item Check” on page 8-38 for programming information and basic examples.

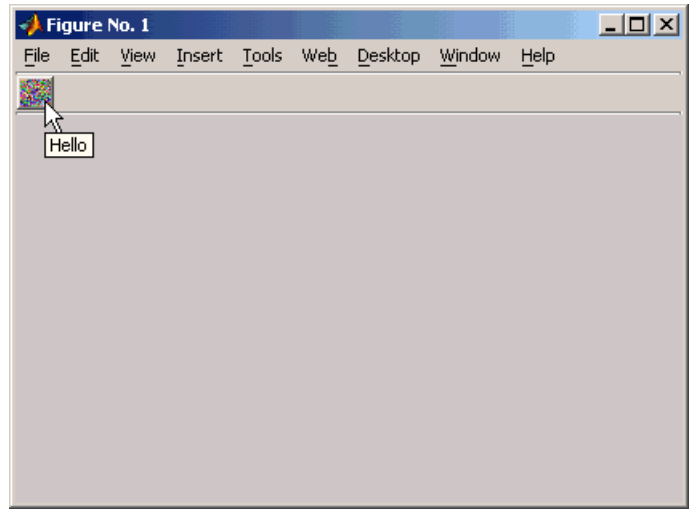
Creating Toolbars

GUIDE provides no tool to enable you do add a toolbar. However, you can add a toolbar by adding code to the opening function.

See “Initialization Callbacks” on page 8-15 for information about the opening function, and the `uitoolbar`, `uipushtool`, and `uitoggletool` reference pages for information and examples.

Add the following code to the GUI’s opening function to produce the toolbar shown. The example creates a toolbar (`uitoolbar`) and places a toggle tool (`uitoggletool`) on it.

```
ht = uitoolbar(hObject)
a(:,:,1) = rand(20);
a(:,:,2) = rand(20);
a(:,:,3) = rand(20);
htt = uitoggletool(ht,'CData',a,'TooltipString','Hello')
```

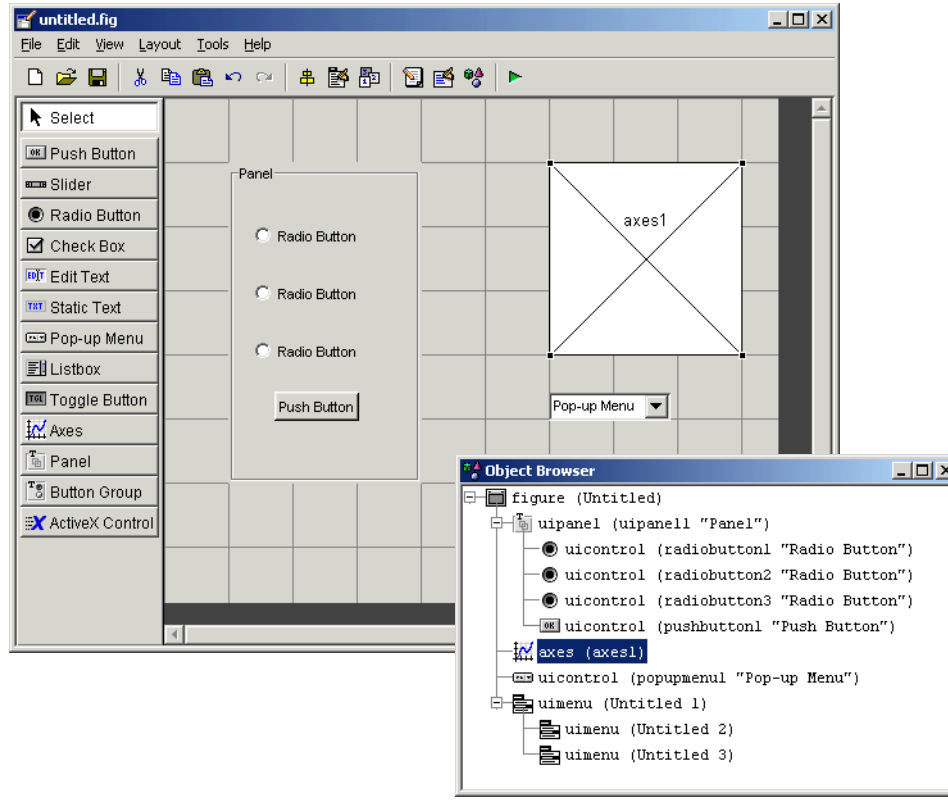


In the opening function, `hObject` is an input argument that holds the figure handle. The `CData` property enables you to display a truecolor image on the toggle tool.

Viewing the Object Hierarchy

The Object Browser displays a hierarchical list of the objects in the figure, including both components and menus. As you lay out your GUI, check the object hierarchy periodically, especially if your GUI contains menus, panels, or button groups.

The following illustration shows a figure object and its child objects. It also shows the child objects of the panel and a menu that was created.



To determine a component's place in the hierarchy, select it in the Layout Editor. It is automatically selected in the Object Browser. Similarly, if you select an object in the Object Browser, it is automatically selected in the Layout Editor.

Designing for Cross-Platform Compatibility

You can create GUIs that behave more consistently when run on different platforms by using:

- "Default System Font" on page 6-63
- "Standard Background Color" on page 6-64
- "Cross-Platform Compatible Units" on page 6-66

Default System Font

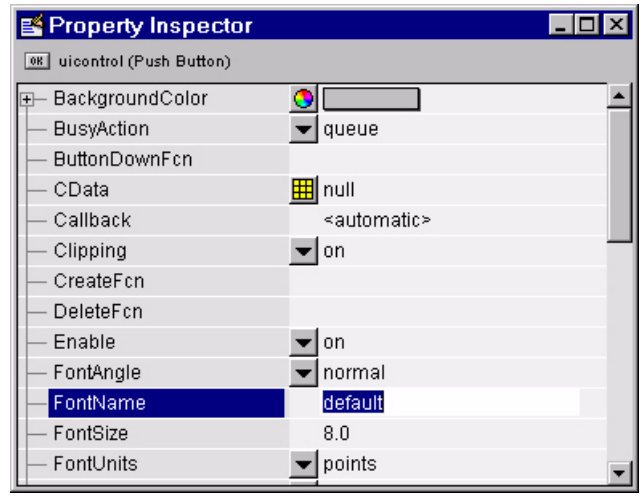
By default, user interface controls (uicontrols) use the default font for the platform on which they are running. For example, when displaying your GUI on PCs, uicontrols uses MS San Serif. When your GUI runs on a different platform, it uses that computer's default font. This provides a consistent look with respect to your GUI and other application GUIs.

If you have set the `FontName` property to a named font and want to return to the default value, you can set the property to the string `default`. This ensures that MATLAB uses the system default at runtime.

From within the GUI M-file, use the set command. For example, if there is a push button in your GUI and its handle is stored in the `pushbutton1` field of the `handles` structure, then the statement,

```
set(handles.pushbutton1,'FontName','default')
```

sets the `FontName` property to use the system default. You can also use the Property Inspector to set this property:



Specifying a Fixed-Width Font

If you want to use a fixed-width font for a user interface control (uicontrol), set its `FontName` property to the string `fixedwidth`. This special identifier ensures that your GUI uses the standard fixed-width font for the target platform.

You can find the name of the fixed-width font that is used on a given platform by querying the root `FixedWidthFontName` property.

```
get(0,'FixedWidthFontName')
```

Using a Specific Font Name

You can specify an actual font name (such as Times or Courier) for the `FontName` property. However, doing so may cause your GUI to look poorly when run on a different computer. If the target computer does not have the specified font, it will substitute another font that may not look good in your GUI or may not be the standard font used for GUIs on that system. Also, different versions of the same named font may have different size requirements for a given set of characters.

Standard Background Color

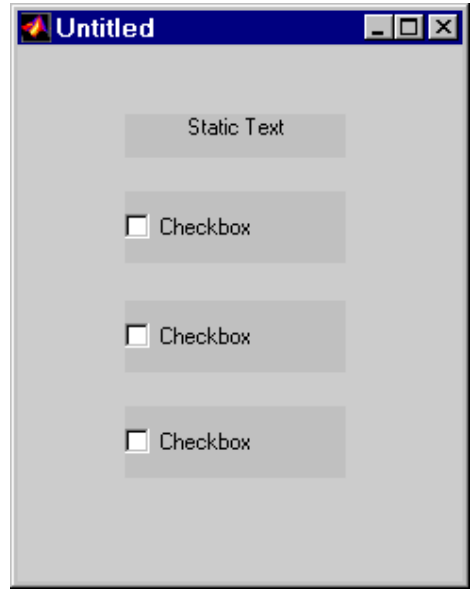
The default component background color is the standard system background color on which the GUI is running. This color varies on different computer

systems, e.g., the standard shade of gray on the PC differs from that on UNIX, and may not match the default GUI background color.

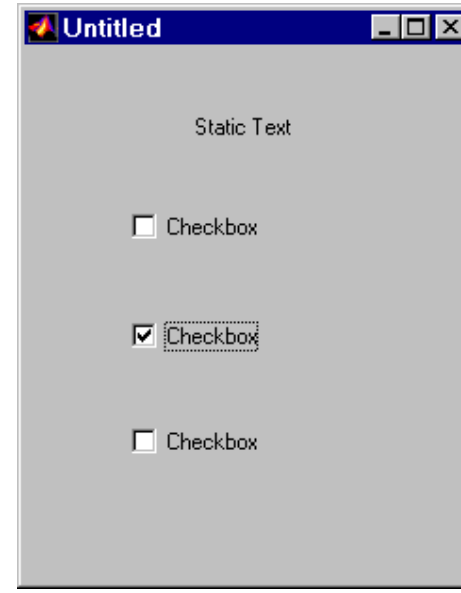
If you use the default component background color, you can use that same color as the background color for your GUI. This provides a consistent look with respect to your GUI and other application GUIs. To do this in GUIDE, check **Options -> Use system color scheme for background** option on the Layout Editor **View** menu.

Note This option is available only if you first select the **Generate FIG-file and M-File** option.

The following figures illustrate the results with and without system color matching.



Without system color matching



With system color matching

Cross-Platform Compatible Units

Cross-platform compatible GUIs should look correct on computers having different screen sizes and resolutions. Since the size of a pixel can vary on different computer displays, using the default figure `Units` of `pixels` does not produce a GUI that looks the same on all platforms.

For this reason, GUIDE defaults the `Units` property for the figure to `characters`.

System-Dependent Units

Character units are defined by characters from the default system font. The width of a character unit equals the width of the letter `x` in the system font. The height of a character unit is the distance between the baselines of two lines of text. Note that character units are not square.

Units and Resize Behavior

If you set your GUI's resize behavior using the GUI options, GUIDE automatically sets the units for the GUI's components in a way that maintains the intended look and feel across platforms. To specify the resize behavior option, select **Options** from the **Tools** menu, then specify **Resize behavior** by selecting **Non-resizable**, **Proportional**, or **Other (Use ResizeFcn)**.

If you choose the **Non-resizable** option, GUIDE defaults the component units to `characters`. If you choose **Proportional**, it defaults the component units to `normalized`. In either case, these settings enable your GUI to automatically adjusts the size and relative spacing of components as the GUI displays on different computers.

If you choose **Other (Use ResizeFcn)**, GUIDE defaults the component units to `characters`. You must provide a `ResizeFcn` callback to customize the GUI's resize behavior.

Note GUIDE does not automatically adjust the component units if you modify the `Resize` property programmatically or in the Property Inspector.

At times, it may be convenient to use a more familiar unit of measure, e.g., inches or centimeters, when you are laying out the GUI. However, to preserve the look of your GUI on different computers, remember to change the figure

`Units` property back to `characters`, and the components' `Units` properties to `characters` (nonresizable GUIs) or `normalized` (resizable GUIs) before you save the GUI.

7

Saving and Running the GUI

Naming a GUI and Its Files (p. 7-2)	Describes the GUI files and how they are named.
Saving a GUI (p. 7-5)	Describes the various ways of saving a GUI in GUIDE.
Running a GUI (p. 7-10)	Tells you how to run a GUI from GUIDE and from the command line.
Saving and Displaying Only the GUI Layout (p. 7-12)	Tells you how to save and display only the inactive GUI layout.

Naming a GUI and Its Files

- "The GUI Files" on page 7-2
- "File and GUI Names" on page 7-3
- "How to Assign Names" on page 7-3
- "Renaming GUIs and GUI Files" on page 7-4

The GUI Files

By default, GUIDE stores a GUI in two files which are generated the first time you save or run the GUI:

- A FIG-file, with extension `.fig`, that contains a complete description of the GUI layout and the GUI components, such as push buttons, axes, panels, menus, and so on. The FIG-file is a binary file and you cannot modify it except by changing the layout in GUIDE. Note that a FIG-file is a kind of MAT-file. See MAT-Files for more information.
- An M-file, with extension .m, that contains the code that controls the GUI, including the callbacks for its components.

These two files usually reside in the same directory. They correspond to the tasks of laying out and programming the GUI. When you lay out the GUI in the Layout Editor, your work is stored in the FIG-file. When you program the GUI, your work is stored in the corresponding M-file.

Note that if your GUI includes ActiveX components, GUIDE also generates a file for each ActiveX component. See "ActiveX Control" on page 8-31 for more information.

Note You can choose to save only the FIG-file for a GUI. See "Saving and Displaying Only the GUI Layout" on page 7-12

For more information about these files, see "GUI Files: An Overview" on page 8-5.

File and GUI Names

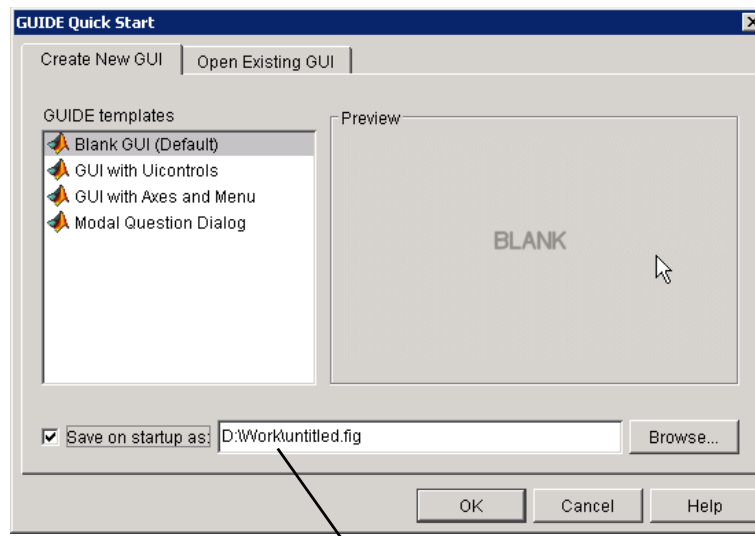
The M-file and the FIG-file that define your GUI must have the same name. This name is also the name of your GUI.

For example, if your files are named `mygui.fig` and `mygui.m`, then the name of the GUI is `mygui`, and you can run the GUI by typing `mygui` at the command line. This assumes that the M-file and FIG-file are in the same directory and that the directory is in your path.

How to Assign Names

You can name the files and the GUI in any of these ways:

- From the GUIDE Quick Start dialog box. Before you select a template, GUIDE lets you select a name for your GUI. If you do this, GUIDE saves the GUI M-file and FIG-file using the name you specify.



- The first time you save the files:
  - Using the Save icon on the Layout Editor toolbar

  - By selecting the **Save** or **Save as** options on the **File** menu

  GUIDE prompts you for a name before saving the GUI.

- The first time you run the GUI:
  - Using the Run icon on the Layout Editor toolbar
  - By selecting **Run** from the **Tools** menu

  GUIDE prompts you for a name before saving and running the GUI.

In all cases, GUIDE creates a template M-file and opens it in your default editor. See “Naming of Callback Functions” on page 8-11 for more information about the template M-file.

Renaming GUIs and GUI Files

To rename a GUI, rename the GUI FIG-file using **Save As** from the Layout Editor **File** menu. When you do this, GUIDE renames both the FIG-file and the GUI M-file, resets any callback properties that contain the old name to use the new name, and updates all instances of the file name in the body of the M-file.

Saving a GUI

This section tells you how to save a GUI you have constructed in GUIDE.

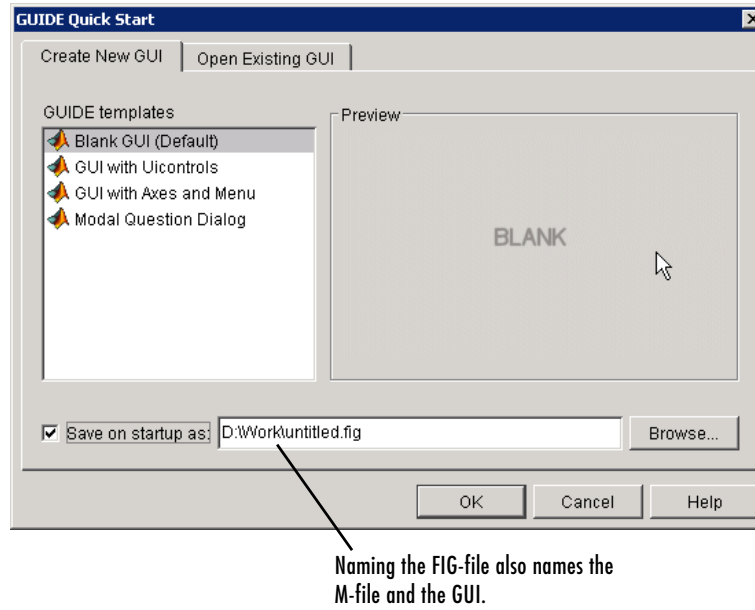
- "Ways to Save a GUI" on page 7-5
- "Saving a New GUI" on page 7-7
- "Saving an Existing GUI" on page 7-9

Note If you want to save or display only a GUI FIG-file, see "Saving and Displaying Only the GUI Layout" on page 7-12.

Ways to Save a GUI

You can save a GUI in GUIDE using any of these actions:

- Selecting a name for the GUI in the GUIDE Quick Start dialog box. When you click **OK**, GUIDE saves the GUI M-file and FIG-file using the name you specify.



- Clicking the Save icon on the Layout Editor toolbar
- Selecting the **Save** or **Save as** options on the **File** menu
- Clicking the Run icon on the Layout Editor toolbar
- Selecting **Run** from the **Tools** menu
- Clicking the **View** button in the Menu Editor

If you choose to run the GUI, GUIDE saves the GUI files before activating the GUI.

In all cases, GUIDE opens the GUI M-file in your default editor.

Note GUI FIG-files that are created or modified with MATLAB 7.0 or a later MATLAB version, are not automatically compatible with Version 6.5 and earlier versions. To make a FIG-file, which is a kind of MAT-file, backward compatible, you must check the MAT-file **Ensure backward compatibility (-v6)** preference in the **Preferences** dialog box under **General** -> **MAT-Files**.

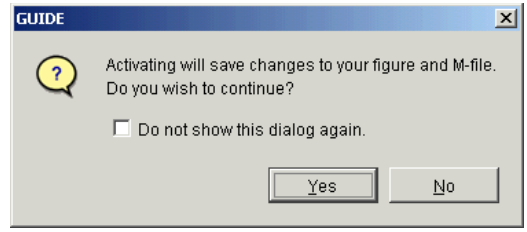
Note that button groups and panels are introduced in MATLAB 7.0, and you should not use them in GUIs that you expect to run in earlier MATLAB versions.

Saving a New GUI

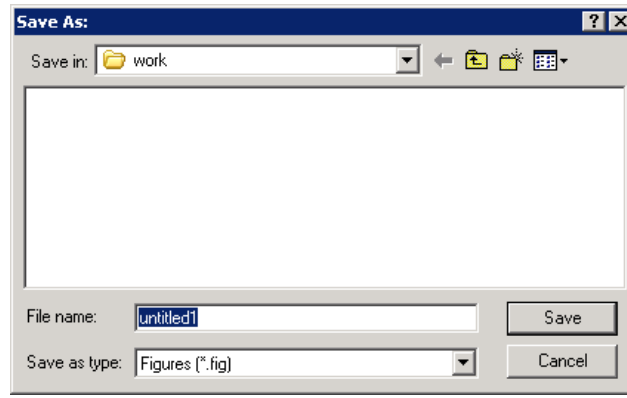
Follow these steps if you are saving a GUI for the first time, or if you are using **Save as** from the **File** menu.

Note When you save a GUI, you can also choose to activate it. If you select **Run** from the **Tools** menu or click the Run icon on the toolbar, GUIDE saves the GUI and also activates it. If you select **Save as** from the **File** menu or click the Save icon on the toolbar, GUIDE saves the GUI without activating it.

1. If you have made changes to the GUI and elect to activate the GUI by selecting **Run** from the **Tools** menu or by clicking the Run icon on the toolbar, GUIDE displays the following dialog box. Click **Yes** to continue.

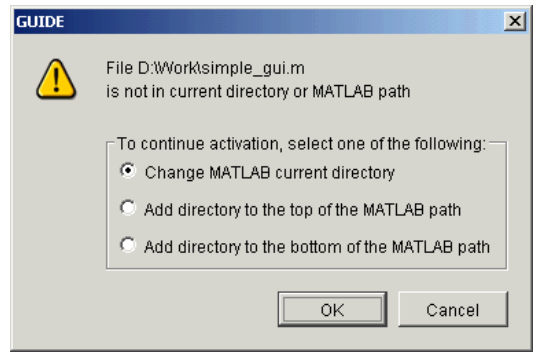


2. If you clicked **Yes** in the previous step, if you are saving the GUI without activating it, or if you are using **Save as** from the File menu, GUIDE opens a **Save As** dialog box in your current directory and prompts you for a FIG-file name.

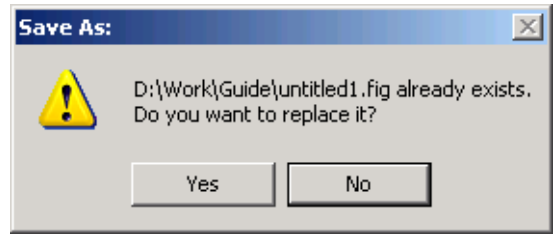


3 Change the directory if you choose, and then enter the name you want to use for the FIG-file. Be sure to choose a writable directory. GUIDE saves both the FIG-file and the M-file using this name.

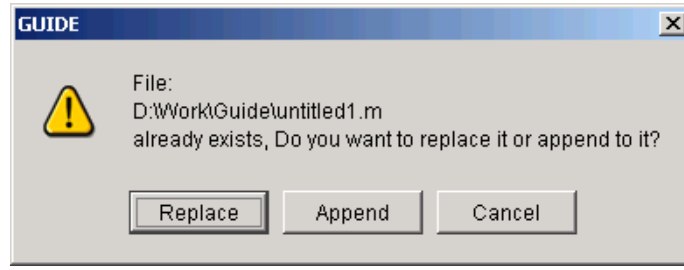
4 If the directory in which you save the GUI is not on the MATLAB path, GUIDE opens a dialog box, giving you the option of changing the current working directory to the directory containing the GUI files, or adding that directory to the top or bottom of the MATLAB path.


GUIDE
File D:\Work\simple_gui.m
is not in current directory or MATLAB path
To continue activation, select one of the following:
Change MATLAB current directory
Add directory to the top of the MATLAB path
Add directory to the bottom of the MATLAB path
OK
Cancel

5 If you choose an existing filename, GUIDE displays a dialog box that asks you if you want to replace the existing FIG-file. Click **Yes** to continue.



6 If you chose **Yes** in the previous step, GUIDE displays a dialog that asks if you want to replace the existing M-file or append to it. If you choose **Append**, GUIDE adds callbacks to the existing M-file for components in the current layout that are not present in the existing M-file. Before you append the new components, ensure that their `Tag` properties do not duplicate `Tag` values that appear in callback function names in the existing M-file. See "Assigning an Identifier to Each Component" on page 6-24 for information about specifying the `Tag` property. See "Naming of Callback Functions" on page 8-11 for more information about callback function names.



7 After you save the files, GUIDE opens the GUI M-file in your default editor. If you elected to run the GUI, it also activates the GUI.

Saving an Existing GUI

Follow these steps if you are saving an existing GUI to its current location. See "Saving a New GUI" on page 7-7 if you are using **Save as** from the **File** menu.

1 If you have made changes to the GUI and choose to activate the GUI by selecting **Run** from the **Tools** menu or by clicking the Run icon on the toolbar, GUIDE displays the following dialog box. Click **Yes** to continue.

Running a GUI

Generally, you run your GUI by executing the M-file that GUIDE generates. This M-file contains the commands to load the GUI and provides a framework for the component callbacks. See “GUI Files: An Overview” on page 8-5 for more information about the M-file.

When you execute the M-file, a fully functional copy of the GUI displays on the screen. You can run a GUI:

- “From the GUIDE Layout Editor” on page 7-10
- “From the Command Line” on page 7-10
- “From an M-file Script” on page 7-11

Note You can display an inactive copy of the GUI figure using the `openfig`, `open`, or `hgload` command. These commands load FIG-files into the MATLAB workspace. Note that the displayed GUI is not active.

From the GUIDE Layout Editor

Run your GUI from the GUIDE layout Editor by:

- Clicking the Run icon on the Layout Editor toolbar
- Selecting **Run** from the **Tools** menu

In either case, GUIDE saves the GUI files before activating the GUI and opens the GUI M-file in your default editor. See “Saving a GUI” on page 7-5 for information about this process. See “GUI Files: An Overview” on page 8-5 for more information about M-files.

From the Command Line

Run your GUI from the command line by entering the name of its M-file. For example, if your GUI M-file is `mygui.m`, type

```
mygui
```

at the command line. If the M-file does not reside on your MATLAB path or in your current MATLAB directory, you must also specify the path.

If a GUI accepts arguments when it is run, they are passed to the GUI's opening function. See "Opening Function" on page 8-15 for more information.

Note Consider whether you want to allow more than one copy of the GUI to be active at the same time. If you want only one GUI to be active, select **Options** from the Layout Editor **View** menu, then select **GUI Allows Only One Instance to Run (Singleton)**. See "GUI Options" on page 5-9 for more information.

From an M-file Script

Run your GUI from an M-file script by executing the GUI M-file. For example, if your GUI M-file is `mygui.m`, include the following statement in your M-file script.

```
mygui
```

The M-file must reside on the MATLAB path or in the current MATLAB directory where the GUI is run.

If a GUI accepts arguments when it is run, they are passed to the GUI's opening function. See "Opening Function" on page 8-15 for more information.

Note Consider whether you want to allow more than one copy of the GUI to be active at the same time. If you want only one GUI to be active, select **Options** from the Layout Editor **View** menu, then select **GUI Allows Only One Instance to Run (Singleton)**. See "GUI Options" on page 5-9 for more information.

Saving and Displaying Only the GUI Layout

- "Saving a GUI Layout Only" on page 7-12
- "Displaying a GUI Layout Only" on page 7-12

Saving a GUI Layout Only

The FIG-file contains the layout of your GUI. You might choose to save only the layout if it is not yet complete and you want to save your work without generating the M-file. To save only the FIG-file for your GUI:

1. Select **Options** from the Layout Editor **View** menu, then select **Generate FIG-file only**. Note that GUIDE can also select this option.
2. Save the GUI using the **Save** or **Save As** item from the **File** menu.

The **Generate FIG-file only** option also allows you to create your own M-file – one that may use a completely different programming style than that provided by the GUIDE M-file.

GUIDE Selects the FIG-File Only Option

GUIDE selects **Generate FIG-file only** as the default if, when you start GUIDE, you provide the name of a FIG-file or the handle of a figure for which a FIG-file exists. For example,

```
guide('mygui.fig')
```

The figure, `mygui.fig`, need not be generated using GUIDE. See the `guide` command for more information.

Displaying a GUI Layout Only

You can display a copy of a GUI figure in the GUIDE Layout Editor using the `openfig`, `open`, or `hgload` command. These commands load FIG-files into the MATLAB workspace. Note that the displayed GUI is not active.

This mode provides limited editing capability and may be useful for GUIs generated in MATLAB versions 5.3 and earlier.

As an example, if your FIG-file is `mygui.fig`, you can display the layout using any of the following commands:

```
open('mygui.fig')

openfig('mygui')

hgload('mygui')
```

You can run a GUI that has no M-file only if you set the `Callback` property of each component in your GUI to a string that MATLAB can evaluate and perform the desired action. This string can be an expression or the name of an M-file.

8

Programming the GUI

Callbacks: An Overview (p. 8-2)	Introduces the functions, referred to as callbacks, that you use to program GUI behavior.
GUI Files: An Overview (p. 8-5)	Describes the files that comprise a GUI and details the structure of the GUI M-file which you must program.
Associating Callbacks with Components (p. 8-8)	Outlines the mechanisms that GUIDE uses for associating a callback with a specific component.
Callback Syntax and Arguments (p. 8-11)	Describes callback naming conventions and input arguments, and introduces the `handles` structure as a tool for communicating among a GUI's callbacks.
Initialization Callbacks (p. 8-15)	Describes the functions, provided by GUIDE, that you can use to initialize a GUI.
Examples: Programming GUI Components (p. 8-19)	Provides a brief example for programming each kind of component.

Callbacks: An Overview

After you have laid out your GUI, you need to program its behavior. The code you write controls how the GUI responds to events such as button clicks, slider movement, menu item selection, or the creation and deletion of components. This programming takes the form of a set of functions, called callbacks, for each component and for the GUI figure itself.

This chapter addresses the programming of GUIs created using GUIDE. In particular, this section explains:

- "What Is a Callback?" on page 8-2
- "Kinds of Callbacks" on page 8-2

What Is a Callback?

A callback is a function that you write and associate with a specific GUI component or with the GUI figure. It controls GUI or component behavior by performing some action in response to an event for its component. This kind of programming is often called event-driven programming.

When an event occurs for a component, MATLAB invokes the component's callback that is triggered by that event. As an example, suppose a GUI has a button that triggers the plotting of some data. When the user clicks the button, MATLAB calls the callback you associated with clicking that button, and the callback, which you have programmed, then gets the data and plots it.

A component can be any control device such as a push button, list box, or slider. It can also be a menu or a container such as a panel or button group. See "Available Components" on page 6-14 for a list and descriptions of components.

Kinds of Callbacks

The GUI figure and each type of component has specific kinds of callbacks with which it can be associated. The callbacks that are available for each component are defined as properties of that component. For example, a push button has five callback properties: `ButtonDownFcn`, `Callback`, `CreateFcn`, `DeleteFcn`, and `KeyPressFcn`. A panel has four callback properties: `ButtonDownFcn`, `CreateFcn`, `DeleteFcn`, and `ResizeFcn`. You can, but are not required to, create a callback function for each of these properties. The GUI itself, which is a figure, also has certain kinds of callbacks with which it can be associated.

Each kind of callback has a triggering mechanism or event that causes it to be called. The following table lists the callback properties that GUIDE makes available, their triggering events, and the components to which they apply.

Callback Property	Triggering Event	Components
`ButtonDownFcn`	Executes when the user presses a mouse button while the pointer is on or within five pixels of a component or figure.	Axes, figure, button group, panel, user interface controls
`Callback`	Control action. Executes, for example, when a user clicks a push button or selects a menu item.	Context menu, menu, user interface controls
`CloseRequestFcn`	Executes when the figure closes.	Figure
`CreateFcn`	Initializes the component when it is created. It executes after the component or figure is created, but before it is displayed.	Axes, figure, button group, context menu, menu, panel, user interface controls
`DeleteFcn`	Performs cleanup operations just before the component or figure is destroyed.	Axes, figure, button group, context menu, menu, panel, user interface controls

Callback Property	Triggering Event	Components
`KeyPressFcn`	Executes when the user presses a keyboard key and the callback's component or figure has focus.	Figure, user interface controls
`ResizeFcn`	Executes when a user resizes a panel, button group, or figure whose figure `Resize` property is set to `On`.	Button group, figure, panel
`SelectionChangeFcn`	Executes when a user selects a different radio button or toggle button in a button group component.	Button group
`WindowButtonDownFcn`	Executes when you press a mouse button while the pointer is in the figure window.	Figure
`WindowButtonMotionFcn`	Executes when you move the pointer within the figure window.	Figure
`WindowButtonUpFcn`	Executes when you release a mouse button.	Figure

Note User interface controls include push buttons, sliders, radio buttons, check boxes, editable text boxes, static text boxes, list boxes, and toggle buttons. They are sometimes referred to as uicontrols.

GUI Files: An Overview

- "M-Files and FIG-Files" on page 8-5
- "GUI M-File Structure" on page 8-6
- "Adding Callback Templates to an Existing GUI M-File" on page 8-7

M-Files and FIG-Files

By default, the first time you save or run a GUI, GUIDE stores the GUI in two files:

- A FIG-file, with extension `.fig`, that contains a complete description of the GUI layout and the GUI components, such as push buttons, axes, panels, menus, and so on. The FIG-file is a binary file and you cannot modify it except by changing the layout in GUIDE. Note that a FIG-file is a kind of MAT-file. See MAT-Files for more information.
- An M-file, with extension `.m`, that initially contains initialization code and templates for some callbacks that are needed to control GUI behavior. You must add the callbacks you write for your GUI components to this file.

  Whenever you save your GUI, GUIDE automatically opens the M-file in your default editor.

The FIG-file and the M-file, usually reside in the same directory. They correspond to the tasks of laying out and programming the GUI. When you lay out the GUI in the Layout Editor, your work is stored in the FIG-file. When you program the GUI, your work is stored in the corresponding M-file.

Note that if your GUI includes ActiveX components, GUIDE also generates a file for each ActiveX component. See "ActiveX Control" on page 8-31 for more information.

For more information about naming and saving a GUI, see Chapter 7, "Saving and Running the GUI". If you want to change the name of your GUI and its files, see "Renaming GUIs and GUI Files" on page 7-4.

Note You can choose to save only the FIG-file for a GUI. See "Saving and Displaying Only the GUI Layout" on page 7-12

GUI M-File Structure

The GUI M-file that GUIDE generates is a function file. The name of the main function is the same as the name of the M-file. For example, if the name of the M-file is `mygui.m`, then the name of the main function is `mygui`. Each callback in the file is a subfunction of the main function.

When GUIDE generates an M-file, it automatically includes templates for the most commonly used callbacks for each component. The M-file also contains initialization code, as well as an opening function callback and an output function callback. You must add code to the component callbacks for your GUI to work as you want. You may also want to add code to the opening function callback and the output function callback. The major sections of the GUI M-file are ordered as shown in the following table.

Section	Description
Comments	Displayed at the command line in response to the `help` command. Edit these as necessary for your GUI.
Initialization	GUIDE initialization tasks. *Do not edit this code.*
Opening function	Performs your initialization tasks before the user has access to the GUI.
Output function	Returns outputs to the MATLAB command line after the opening function returns control and before control returns to the command line.
Component and figure callbacks	Control the behavior of the GUI figure and of individual components. MATLAB calls a callback in response to a particular event for a component or for the figure itself.

Adding Callback Templates to an Existing GUI M-File

When you save the GUI, GUIDE automatically adds templates for some callbacks to the M-file. However, you may want to add other callbacks to the M-file.

Within GUIDE, you can add a callback subfunction template to the GUI M-file in one of two ways. With the component selected for which you want to add the callback:

- Click the right mouse button to display the Layout Editor context menu. Select the desired callback from the **View callbacks** submenu. GUIDE adds the callback template to the GUI M-file and opens the M-file for editing at the callback it just added.
- In the View menu, select the desired callback from the **View callbacks** submenu. GUIDE adds the callback template to the GUI M-file and opens the M-file for editing at the callback you just added.

Note In either case, if you select a callback that already exists in the GUI M-file, GUIDE adds no callback, but opens the M-file for editing at the callback you select.

For more information, see “Associating Callbacks with Components” on page 8-8.

Associating Callbacks with Components

A GUI can have many components and GUIDE provides a way of specifying which callback should run in response to a particular event for a particular component. The callback that runs when the user clicks a **Yes** button is not the one that runs for the **No** button. Each menu item usually performs a different function.

GUIDE uses each component's callback properties to associate specific callbacks with that component. This topic tells you how GUIDE can automatically set the values of callback properties.

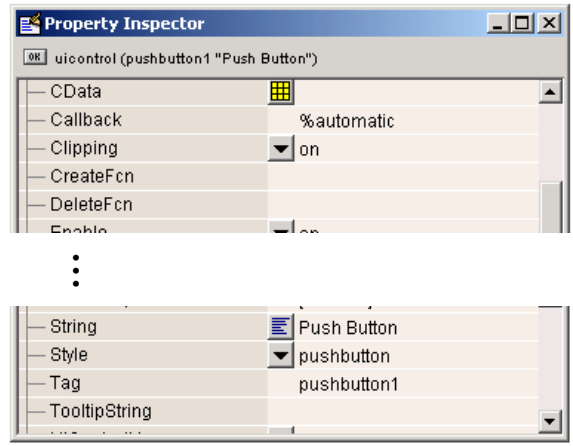
Note "Kinds of Callbacks" on page 8-2 provides a list of callback properties and the components to which each applies.

Setting Callback Properties Automatically

GUIDE initially sets the value of the most commonly used callback properties for each component to `%automatic`. For example, a push button has five callback properties, `ButtonDownFcn`, `Callback`, `CreateFcn`, `DeleteFcn`, and `KeyPressFcn`. GUIDE sets only the `Callback` property, the most commonly used callback, to `%automatic`. You can use the Property Inspector to set the other callback properties to `%automatic`.

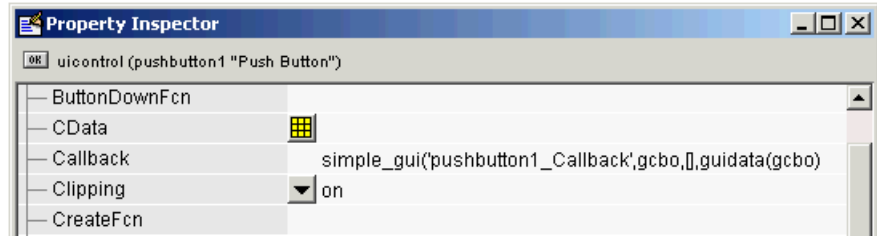
When you next save the GUI, GUIDE replaces `%automatic` with a MATLAB expression that is the GUI calling sequence for the callback. Within the calling sequence, it constructs the callback name, i.e., the subfunction name, from the component's `Tag` property and the name of the callback property. The following figure shows an example of a push button's `Callback` and `Tag` properties in the GUIDE Property Inspector before the GUI is saved.

Note If you change the string `%automatic` before saving the GUI, GUIDE does not automatically add a callback for that component or menu item.



When you save the GUI, GUIDE constructs the name of the callback by appending an underscore (_) and the name of the callback property to the value of the component's `Tag` property. For example, the MATLAB expression for the `Callback` property for a push button in the GUI `simple_gui` with `Tag` property `pushbutton1` is

```
simple_gui(pushbutton1_Callback,gcbo,[],guidata(gcbo))
```



`simple_gui` is the name of the GUI M-file as well as the name of the main function for that GUI. The remaining arguments generate input arguments for `pushbutton1_Callback`. Specifically,

- `gcbo` is a command that returns the handle of the callback object (i.e., `pushbutton1`).
- `[]` is a place holder for the currently unused `eventdata` argument.
- `guidata(gcbo)` returns the `handles` structure for this GUI.

See “Input Arguments” on page 8-13 for information about the callback input arguments.

When you save the GUI, GUIDE also opens the GUI M-file in your editor. The M-file then contains a template for the `Callback` callback for the component whose `Tag` is `pushbutton1`. If you activate the GUI, pressing the push button triggers the execution of the `Callback` callback for the component.

For information about changing the callback name after GUIDE assigns it, see “Changing Callback Names Assigned by GUIDE” on page 8-12.

The next topic, “Callback Syntax and Arguments” on page 8-11, provides more information about the callback template.

Callback Syntax and Arguments

GUIDE defines conventions for callback syntax and arguments and implements these conventions in the callback templates it adds to the M-file. Each template is similar to this one for the `Callback` subfunction for a push button.

```
% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles) %#ok
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

...
```

The first comment line describes the event that triggers execution of the callback. This is followed by the function definition line. The remaining comments describe the input arguments.

Insert your code after the last comment.

Note You can avoid automatic generation of the callback comment lines for new callbacks. In the **Preferences** dialog box, select **GUIDE** and uncheck **Add comments for newly generated callback functions**.

This topic discusses these aspects of the template:

- "Naming of Callback Functions" on page 8-11
- "Changing Callback Names Assigned by GUIDE" on page 8-12
- "Input Arguments" on page 8-13
- "The Handles Structure" on page 8-13

Naming of Callback Functions

The callback example above shows the following function definition:

```
function pushbutton1_Callback(hObject,eventdata,handles)
```

When GUIDE generates the template, it creates the callback name by appending an underscore (_) and the name of the callback property to the component's `Tag` property. In the example above, `pushbutton1` is the `Tag` property for the push button, and `Callback` is one of the push button's callback properties. The `Tag` property uniquely identifies a component within the GUI.

The first time you save the GUI after adding a component, GUIDE adds callbacks for that component to the M-file and generates the callback names using the current value of the `Tag` property. If you want to change the default `Tag` value, you should do it before you save the GUI.

See "Associating Callbacks with Components" on page 8-8 for more information.

Changing Callback Names Assigned by GUIDE

You can change callback names assigned by GUIDE in either of the following ways:

- "Changing the Tag Property" on page 8-12
- "Changing the Callback Property" on page 8-13

Note If possible, change callback names for a component immediately after you add the component to the layout and before you save the GUI.

Changing the Tag Property

You can change `Tag` properties to give a component's callbacks a more meaningful name, e.g., you might change the `Tag` property from `pushbutton1` to `closebutton`. If possible, change the `Tag` property before saving the GUI, then GUIDE automatically uses the new value when it names the callbacks. However, if you change the `Tag` property after saving the GUI, GUIDE updates the following items according to the new `Tag`, provided that all components have distinct tags:

- The component's callback functions in the M-file
- The value of the component's callback properties, which you can view in the Property Inspector

- References in the M-file to the field of the `handles` structure that contains the component's handle. See "The Handles Structure" on page 8-13 for more information about the `handles` structure.

Changing the Callback Property

To rename a particular callback subfunction without changing the `Tag` property,

- Replace the name string in the callback property with the new name. For example, if the value of the callback property for a push button in `mygui` is

  ```
  mygui('pushbutton1_Callback',gcbo,[],guidata(gcbo))
  ```

  the string `pushbutton1_Callback` is the name of the callback function. Change the name to the desired name, for example, `closebutton_Callback`.
- You must update all instances of the callback function name in the M-file.
- You must update references in the M-file to the field of the `handles` structure that contains the component's handle. See "The Handles Structure" on page 8-13 for more information about the `handles` structure.

Input Arguments

All callbacks in the GUI M-file have the following input arguments:

- `hObject` — Handle of the object, e.g., the component, for which the callback was triggered.
- `eventdata` — Reserved for later use.
- `handles` — Structure that contains the handles of all the objects in the figure. It may also contain application-defined data. See "The Handles Structure" on page 8-13 for information about this structure.

The Handles Structure

GUIDE creates a `handles` structure that contains the handles of all the objects in the figure. For a GUI that contains an edit text, a panel, a pop-up menu, and a push button, the `handles` structure originally looks similar to this. GUIDE uses each component's `Tag` property to name the structure element for its handle.

```
handles =
        figure1: 160.0011
          edit1: 9.0020
        uipanel1: 8.0017
     popupmenu1: 7.0018
    pushbutton1: 161.0011
         output: 160.0011
```

GUIDE creates and maintains the `handles` structure as GUI data. It is passed as an input argument to all callbacks and enables a GUIs callbacks to share property values and application data.

For information about GUI data, see “Mechanisms for Managing Data” on page 9-2 and the `guidata` reference page.

For information about adding fields to the `handles` structure and instructions for correctly saving the structure, see Chapter 9, “Managing Application-Defined Data.”

Initialization Callbacks

GUIDE automatically includes two callbacks, the opening function and the output function, in every GUI M-file it creates.

- "Opening Function" on page 8-15
- "Output Function" on page 8-17

For information about programming component callbacks, see "Examples: Programming GUI Components" on page 8-19.

Opening Function

The opening function is the first callback in every GUI M-file. It is executed just before the GUI is made visible to the user, but after all the components have been created, i.e., after the components' `CreateFcn` callbacks, if any, have been run.

You can use the opening function to perform your initialization tasks before the user has access to the GUI. For example, you can use it to create data or to read data from an external source. GUI command line arguments are passed to the opening function.

- "Function Naming and Template" on page 8-15
- "Input Arguments" on page 8-16
- "Initial Template Code" on page 8-16

Function Naming and Template

GUIDE names the opening function by appending `_OpeningFcn` to the name of the M-file. This is an example of an opening function template as it might appear in the `mygui` M-file.

```
% --- Executes just before mygui is made visible.
function mygui_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to mygui (see VARARGIN)
```

```
% Choose default command line output for mygui
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes mygui wait for user response (see UIRESUME)
% uiwait(handles.mygui);
```

Input Arguments

The opening function has four input arguments `hObject`, `eventdata`, `handles`, and `varargin`. The first three are the same as described in "Input Arguments" on page 8-13. the last argument, `varargin`, enables you to pass arguments from the command line to the opening function. The opening function can make such arguments available to the callbacks by adding them to the `handles` structure.

For more information about `varargin`, see the `varargin` reference page and Passing Variable Numbers of Arguments in the MATLAB documentation.

All command line arguments are passed to the opening function via `varargin`. If you open the GUI with a property name/property value pair as arguments, the GUI opens with the property set to the specified value. For example, `my_gui('Position', [71.8 44.9 74.8 19.7])` opens the GUI at the specified position, since `Position` is a valid figure property.

If the input argument is not a valid figure property, you must add code to the opening function to make use of the argument. For an example, look at the opening function for the **Modal Question Dialog** GUI template, available from the GUIDE Quick Start dialog box. The added code enables you to open the modal dialog with the syntax

```
mygui('String','Do you want to exit?')
```

which displays the text `'Do you want to exit?'` on the GUI. In this case, it is necessary to add code to the opening function because `'String'` is not a valid figure property.

Initial Template Code

Initially, the input function template contains these lines of code:

- `handles.output = hObject` adds a new element, `output`, to the `handles` structure and assigns it the value of the input argument `hObject`, which is the handle of the figure, i.e., the handle of the GUI. This handle is used later by the output function. For more information about the output function, see "Output Function" on page 8-17.
- `guidata(hObject,handles)` saves the `handles` structure. You must use `guidata` to save any changes that you make to the `handles` structure. It is not sufficient just to set the value of a `handles` field. See "The Handles Structure" on page 8-13 and "GUI Data" on page 9-2 or more information.
- `uiwait(handles.mygui)`, initially commented out, blocks GUI execution until `uiresume` is called or the GUI is deleted. Note that `uiwait` allows the user access to other MATLAB windows. Remove the comment symbol for this statement if you want the GUI to be blocking when it opens.

Output Function

The output function returns, to the command line, outputs that are generated during its execution. It is executed when the opening function returns control and before control returns to the command line. This means that you must generate the outputs in the opening function, or call `uiwait` in the opening function to pause its execution while other callbacks generate outputs.

- "Function Naming and Template" on page 8-17
- "Input Arguments" on page 8-18
- "Output Arguments" on page 8-18

Function Naming and Template

GUIDE names the output function by appending `_OutputFcn` to the name of the M-file. This is an example of an output function template as it might appear in the `mygui` M-file.

```
% --- Outputs from this function are returned to the command line.
function varargout = mygui_OutputFcn(hObject, eventdata,...
                     handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

```
% Get default command line output from handles structure
varargout{1} = handles.output;
```

Input Arguments

The output function has three input arguments: `hObject`, `eventdata`, and `handles`. They are the same as described in "Input Arguments" on page 8-13.

Output Arguments

The output function has one output argument, `varargout`, which it returns to the command line. By default, the output function assigns `handles.output` to `varargout`. So the default output is the handle to the GUI, which was assigned to `handles.output` in the opening function.

You can change the output by:

- Changing the value of `handles.output`. It can be any valid MATLAB value including a structure or cell array.
- Adding output arguments to `varargout`.

  `varargout` is a cell array. It can contain any number of output arguments. By default, GUIDE creates just one output argument, `handles.output`. To create an additional output argument, create a new field in the `handles` structure and add it to `varargout` using a command similar to

  ```
  varargout{2} = handles.second_output;
  ```

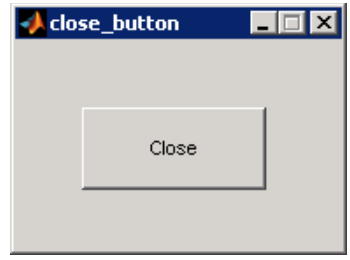
Examples: Programming GUI Components

This section provides basic code examples to get you started programming the most commonly used components.

- "Push Button" on page 8-19
- "Toggle Button" on page 8-20
- "Radio Button" on page 8-21
- "Check Box" on page 8-21
- "Edit Text" on page 8-22
- "Slider" on page 8-23
- "List Box" on page 8-23
- "Pop-Up Menu" on page 8-24
- "Panel" on page 8-25
- "Button Group" on page 8-26
- "Axes" on page 8-27
- "ActiveX Control" on page 8-31
- "Menu Item" on page 8-37

Push Button

This example contains only a push button. Clicking the button, closes the GUI.



This is the push button's `Callback` callback. It displays the string `Goodbye` at the command line and then closes the GUI.

```
function pushbutton1_Callback(hObject, eventdata, handles)
display Goodbye
delete(handles.figure1);
```

Adding an Image to a Push Button or Toggle Button

To add an image to a push button or toggle button, assign the button's `CData` property an m-by-n-by-3 array of RGB values that defines a truecolor image. For example, the array a defines 16-by-64 truecolor image using random values between 0 and 1 (generated by `rand`).

```
a(:,:,1) = rand(16,64);
a(:,:,2) = rand(16,64);
a(:,:,3) = rand(16,64);
set(hObject,'CData',a)
```

To add the image when the button is created, add the code to the button's `CreateFcn` callback. You may want to delete the value of the button's `String` property, which would usually be used as a label.

See `ind2rgb` for information on converting a matrix `X` and corresponding colormap map, i.e., an `(X, MAP)` image, to RGB (truecolor) format.

Toggle Button

The callback for a toggle button needs to query the toggle button to determine what state it is in. MATLAB sets the `Value` property equal to the `Max` property when the toggle button is pressed (`Max` is 1 by default) and equal to the `Min` property when the toggle button is not pressed (`Min` is 0 by default).

The following code illustrates how to program the callback in the GUI M-file.

```
function togglebutton1_Callback(hObject, eventdata, handles)
button_state = get(hObject,'Value');
if button_state == get(hObject,'Max')
   % Toggle button is pressed-take approperiate action
   ...
```

```
elseif button_state == get(hObject,'Min')
   % Toggle button is not pressed-take appropriate action
    ...
end
```

Note Use a button group to manage exclusive selection behavior for toggle buttons. You must include the code to control these toggle buttons in the button group's `SelectionChangeFcn` callback function, not in the individual toggle button `Callback` functions. A button group overwrites the `Callback` properties of radio buttons and toggle buttons that it manages. See "Button Group" on page 8-26 for more information.

Radio Button

You can determine the current state of a radio button from within its callback by querying the state of its `Value` property, as illustrated in the following example:

```
if (get(hObject,'Value') == get(hObject,'Max'))
   % Radio button is selected-take approriate action
else
   % Radio button is not selected-take approriate action
end
```

Note Use a button group to manage exclusive selection behavior for radio buttons. You must include the code to control these radio buttons in the button group's `SelectionChangeFcn` callback function, not in the individual radio button `Callback` functions. A button group overwrites the `Callback` properties of radio buttons and toggle buttons that it manages. See "Button Group" on page 8-26 for more information.

Check Box

You can determine the current state of a check box from within its callback by querying the state of its `Value` property, as illustrated in the following example:

```
function checkbox1_Callback(hObject, eventdata, handles)
if (get(hObject,'Value') == get(hObject,'Max'))
   % Checkbox is checked-take approriate action
else
   % Checkbox is not checked-take approriate action
end
```

Edit Text

To obtain the string a user typed in an edit box, get the `String` property in the the `Callback` callback.

```
function edittext1_Callback(hObject, eventdata, handles)
user_string = get(hObject,'String');
% Proceed with callback
```

If the edit text `Max` and `Min` properties are set such that `Max - Min > 1`, the user can enter multiple lines. For example, setting `Max` to `2`, with the default value of `0` for `Min`, enables users to enter multiple lines.

Retrieving Numeric Data from an Edit Text Component

MATLAB returns the value of the edit text `String` property as a character string. If you want users to enter numeric values, you must convert the characters to numbers. You can do this using the `str2double` command, which converts strings to doubles. If the user enters nonnumeric characters, `str2double` returns `NaN`.

You can use the following code in the edit text callback. It gets the value of the `String` property and converts it to a double. It then checks whether the converted value is `NaN` (`isnan`), indicating the user entered a nonnumeric character and displays an error dialog (`errordlg`).

```
function edittext1_Callback(hObject, eventdata, handles)
user_entry = str2double(get(hObject,'string'));
if isnan(user_entry)
   errordlg('You must enter a numeric value','Bad Input','modal')
end
% Proceed with callback...
```

Triggering Callback Execution

For both UNIX and Windows, clicking inside the GUI but outside the edit text causes the edit text callback to execute. The user can also press **Enter** for an edit text that allows only a single line of text, or **Ctrl+Enter** for an edit text that allows multiple lines.

Available Keyboard Accelerators

GUI users can use the following keyboard accelerators to modify the content of an edit text. These accelerators are not modifiable.

- Ctrl+X – Cut
- Ctrl+C – Copy
- Ctrl+V – Paste
- Ctrl+H – Delete last character
- Ctrl+A – Select all

Slider

You can determine the current value of a slider from within its callback by querying its `Value` property, as illustrated in the following example:

```
function slider1_Callback(hObject, eventdata, handles)
slider_value = get(hObject,'Value');
% Proceed with callback...
```

The `Max` and `Min` properties specify the slider's maximum and minimum values. The slider's range is `Max - Min`.

List Box

When the list box `Callback` callback is triggered, the list box `Value` property contains the index of the selected item, where `1` corresponds to the first item in the list. The `String` property contains the list as a cell array of strings.

This example retrieves the selected string. It assumes `listbox1` is the value of the `Tag` property. Note that it is necessary to convert the value returned from the `String` property from a cell array to a string.

```
function listbox1_Callback(hObject, eventdata, handles)
index_selected = get(handles.listbox1,'Value');
list = get(handles.listbox1,'String');
item_selected = list{index_selected}; % Convert from cell array
                                      % to string
```

Triggering Callback Execution

MATLAB executes the list box's Callback callback after the mouse button is released or after certain key press events:

- The arrow keys change the `Value` property and trigger callback execution.
- **Enter** and **Space** do not change the `Value` property but trigger callback execution.

If the user double-clicks, the callback executes after each click. MATLAB sets the figure's `SelectionType` property to `normal` on the first click and to `open` on the second click. The callback can query the `SelectionType` property to determine if it was a single- or double-click.

List Box Examples

See the following examples for more information on using list boxes:

- "List Box Directory Reader" on page 10-9 — Shows how to creates a GUI that displays the contents of directories in a list box and enables users to open a variety of file types by double-clicking on the filename.
- "Accessing Workspace Variables from a List Box" on page 10-16 — Shows how to access variables in the MATLAB base workspace from a list box GUI.

Pop-Up Menu

When the pop-up menu `Callback` callback is triggered, the pop-up menu `Value` property contains the index of the selected item, where `1` corresponds to the first item on the menu. The `String` property contains the menu items as a cell array of strings.

Using Only the Index of the Selected Menu Item

This example retrieves only the index of the item selected. It uses a switch statement to take action based on the value. If the contents of the pop-up menu

is fixed, then you can use this approach. Else, you can use the index to retrieve the actual string for the selected item.

```
function popupmenu1_Callback(hObject, eventdata, handles)
val = get(hObject,'Value');
switch val
case 1
% User selected the first item
case 2
% User selected the second item
% Proceed with callback...
```

Using the Index to Determine the Selected String

This example retrieves the actual string selected in the pop-up menu. It uses the pop-up menu `Value` property to index into the list of strings. This approach may be useful if your program dynamically loads the contents of the pop-up menu based on user action and you need to obtain the selected string. Note that it is necessary to convert the value returned by the `String` property from a cell array to a string.

```
function popupmenu1_Callback(hObject, eventdata, handles)
val = get(hObject,'Value');
string_list = get(hObject,'String');
selected_string = string_list{val}; % Convert from cell array
                                    % to string
% Proceed with callback...
```

Panel

Panels group GUI components and can make a GUI easier to understand by visually grouping related controls. A panel can contain panels and button groups as well as axes and user interface controls such as push buttons, sliders, pop-up menus, etc. The position of each component within a panel is interpreted relative to the lower left corner of the panel.

If the GUI is resized, the panel and its components are also resized, but you may want to adjust the size and position of the panel's components. You can do this if you set the GUI **Resize behavior** to **Other (Use ResizeFcn)** and provide a `ResizeFcn` callback for the panel.

Note To set **Resize behavior** for the figure to **Other (Use ResizeFcn)**, select **GUI Options** from the Layout Editor **Tools** menu.

1 This example of a `ResizeFcn` callback retrieves the handles of its children from its `Children` property.

   The order in which components appear in a child list is the reverse of the order in which they tab. See "Setting Tab Order" on page 6-45 for information about using the Tab Order Editor to determine the tab order.

2 It then modifies the `Position` properties of the child components relative to the panel's new size. The `Position` property is a four-element vector that specifies the size of the component and its location with respect to the bottom left corner of its parent. The vector is of the form:

   ```
   [left bottom width height]
   ```

   The example assumes the panel `Units` property is `normalized`. That is, units map the lower-left corner of the panel to (0,0) and the upper-right corner to (1.0,1.0). Because of this, the example can modify the component size relative to the panel size without knowing the actual size of the panel. It positions the left edge of a button 20% of the panel's width from the left edge of the panel and its bottom edge 40% of the panel's height from the bottom of the panel. It sets the button's height to be 50% of the panel's width and 20% of its height.

   ```
   function uipanel1_ResizeFcn(hObject, eventdata, handles)
   children = get(hObject,'Children');
   child_position = get(children(1);'Position');
   child_position = [.20,.40,.50,.20];
   set(children(1),'Position',child_position);
   % Proceed with callback...
   ```

Button Group

Button groups are like panels, but can be used to manage exclusive selection behavior for radio buttons and toggle buttons. The button group's `SelectionChangeFcn` callback is called whenever a selection is made.

For radio buttons and toggle buttons that are managed by a button group, you must include the code to control them in the button group's `SelectionChangeFcn` callback function, not in the individual component `Callback` functions. A button group overwrites the `Callback` properties of radio buttons and toggle buttons that it manages.

This example of a `SelectionChangeFcn` callback uses the `Tag` property of the selected object to choose the appropriate code to execute.

```
function uibuttongroup1_SelectionChangeFcn(hObject,eventdata,handles)
switch get(hObject,'Tag')   % Get Tag of selected object
    case 'radiobutton1'
        % Code for when radiobutton1 is selected goes here.
    case 'radiobutton2'
        % Code for when radiobutton2 is selected goes here.
    % Continue with more cases as necessary.
end
```

Note For a button group, contrary to the automatically generated comment, `hObject` is the handle of the selected radio button or toggle button, and not the handle of the button group. Use `handles.uipanel`*n* as the handle of the button group, where `uipanel`*n* is the `Tag` of the button group.

The code above refers to the handle of `uipanel1` rather than the handle of `uibuttongroup1`, because a button group is a kind of panel.

Axes

Axes components enable your GUI to display graphics, such as graphs and images. This topic provides two examples:

- "Plotting to a GUI with a Single Axes" on page 8-27
- "Plotting to a GUI with Multiple Axes" on page 8-29

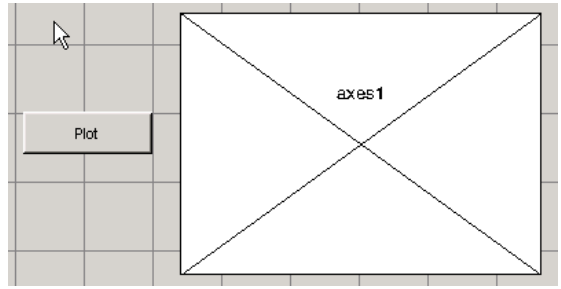
Plotting to a GUI with a Single Axes

If a GUI contains only one axes, MATLAB automatically generates the plot in that axes.

In most cases, you create a plot in an axes from a callback that belongs to some other component in the GUI. For example, pressing a button might trigger the

plotting of a graph to an axes. In this case, the button's `Callback` callback contains the code that generates the plot.

This example plots a graph in an axes when a button is pressed. The following figure shows a push button and an axes as they might appear in the Layout Editor.

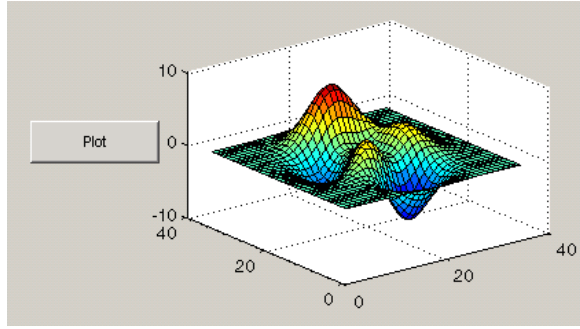


1. Add this code to the push button's `Callback` callback. The `surf` function produces a 3-D shaded surface plot. `peaks` is a function of two variables, obtained by translating and scaling Gaussian distributions.

   ```
   surf(peaks(35));
   ```

2. Run the GUI by selecting **Run** from the **Tools** menu.

3. Click the **Plot** button. The plot displays in the axes.

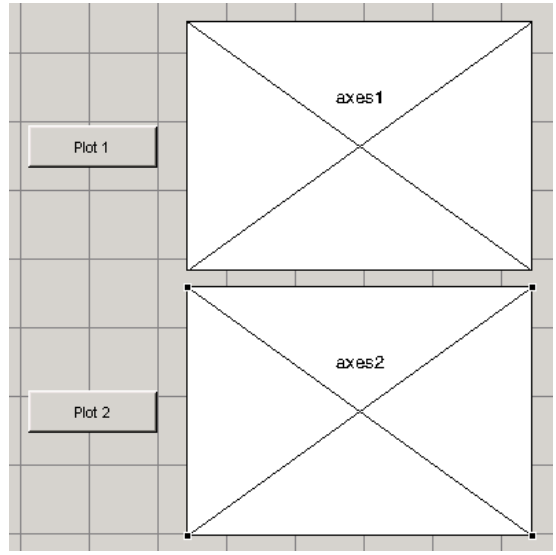


Plotting to a GUI with Multiple Axes

If a GUI has multiple axes, explicitly specify which axes you want to target before you issue a plotting command. You can make the axes you want to target the current axes by using the `axes` function. In the following command, `axes1` is the `Tag` property of the target axes.

```
axes(handles.axes1)
```

The following example contains two axes and two buttons. Clicking one button generates a plot in one axes and clicking the other button generates a plot in the other axes. The following figure shows these components as they might appear in the Layout Editor.



1. Add this code to the **Plot 1** push button's `Callback` callback. The `surf` function produces a 3-D shaded surface plot. `peaks` is a function of two variables, obtained by translating and scaling Gaussian distributions.

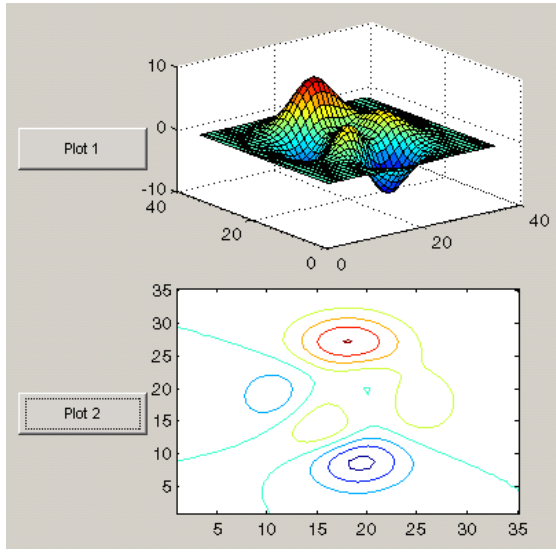
   ```
   axes(handles.axes1)
   surf(peaks(35));
   ```

2 Add this code to the **Plot 2** push button's `Callback` callback. The `contour` function displays the contour plot of a matrix, in this case the output of `peaks`.

```
axes(handles.axes2)
contour(peaks(35));
```

3 Run the GUI by selecting **Run** from the **Tools** menu.

4 Click the **Plot 1** button to display the surf plot in the first axes. Click the **Plot 2** button to display the contour plot in the second axes.



See "GUI with Multiple Axes" on page 10-2 for a more complex example that uses two axes.

Note See "Axes Properties" in the MATLAB documentation for information about properties that you can set to control many aspects of axes behavior and appearance.

If your GUI contains axes, you should make sure that the **Command-line accessibility** option in the **GUI Options** dialog box is set to **Callback** (the

default). From the Layout Editor select **Tools -> GUI Options -> Command Line Accessibility: Callback**. See “Command-Line Accessibility” on page 5-10 for more information about how this option works.

ActiveX Control

This example programs a sample ActiveX control **Mwsamp Control**. It first enables a user to change the radius of a circle by clicking on the circle. It then programs a slider on the GUI to do the same thing.

- “Programming an ActiveX Control” on page 8-31
- “Programming a User Interface Control to Update an ActiveX Control” on page 8-34

This topic also discusses:

- “Viewing the Methods for an ActiveX Control” on page 8-35
- “Saving a GUI that Contains an ActiveX Control” on page 8-36
- “Compiling a GUI that Contains an ActiveX Control” on page 8-37

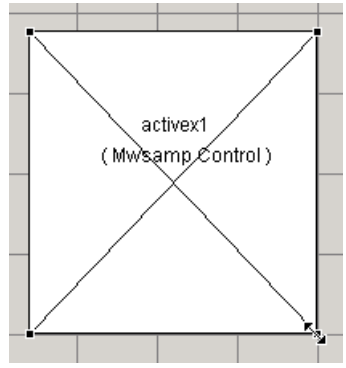
Note See MATLAB COM Client Support in the online MATLAB documentation to learn more about ActiveX controls.

Programming an ActiveX Control

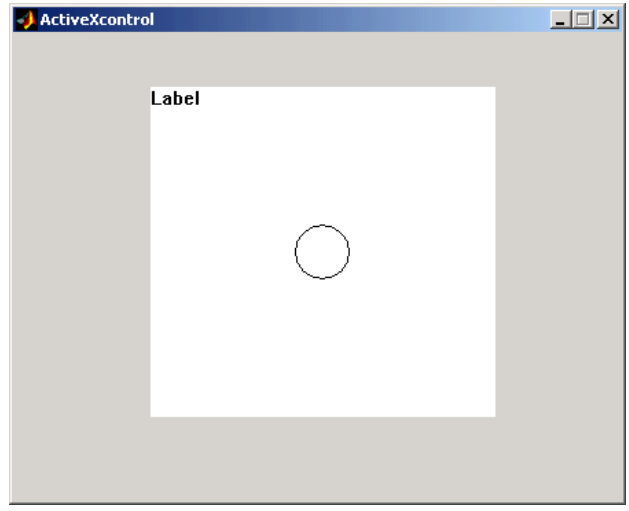
The sample ActiveX control **Mwsamp Control** contains a circle in the middle of a square**.** This example programs the control to change the circle radius when the user clicks the circle, and to update the label to display the new radius.

1. Add the sample ActiveX control **Mwsamp** to your GUI and resize it to approximately the size of the square shown in the preview pane. The following figure shows the ActiveX control as it appears in the Layout Editor.

   If you need help adding the component, see “Adding Components to the Layout Area” on page 6-18.

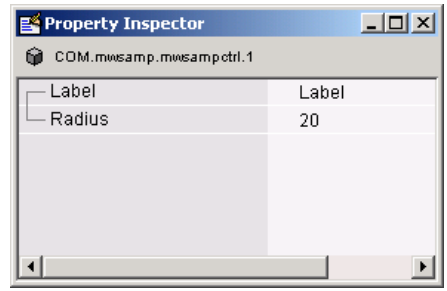


2 Activate the GUI by clicking the Run icon on the toolbar and save the GUI when prompted. GUIDE displays the GUI shown in the following figure and opens the GUI M-file.



3 View the ActiveX Properties with the Property Inspector. Select the control in the Layout Editor, and then select **Property Inspector** from the **View** menu or by clicking the Property Inspector icon .

The following figure shows the properties of the `mwsamp` ActiveX control as they appear in the Property Inspector.



The ActiveX control `mwsamp` has just two properties:

- `Label`, which contains the text that appears at the top of the control
- `Radius`, the default radius of the circle, which is 20

4 Add the following code to the `mswamp` control's `Click` callback. This code programs the ActiveX control to change the circle radius when the user clicks the circle, and updates the label to display the new radius.

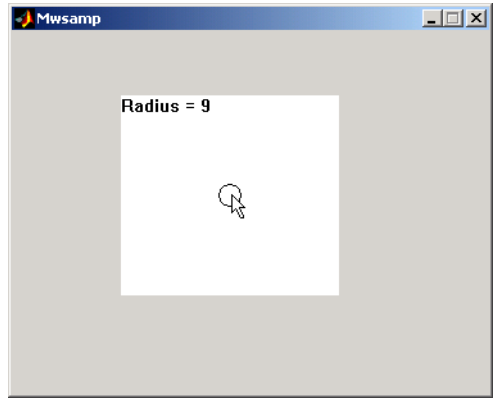
```
hObject.radius = .9*hObject.radius;
hObject.label = ['Radius = ' num2str(hObject.radius)];
refresh(handles.figure1);
```

To locate the `Click` callback in the GUI M-file, select **View Callbacks** from the **View** menu and then select **Click**.

5 Add the following commands to the opening function. This code initializes the label when you first open the GUI.

```
handles.activex1.label = ...
['Radius = ' num2str(handles.activex1.radius)];
```

Now, when you open the GUI and click the ActiveX control, the radius of the circle is reduced by 10 percent and the new value of the radius is displayed. The following figure shows the GUI after clicking the circle six times.



If you click the GUI enough times, the circle disappears entirely.

Programming a User Interface Control to Update an ActiveX Control

This topic continues the previous example by adding a slider to the GUI and programming the slider to change the circle radius. This example must also update the slider if the user clicks on the circle.

1. Add a slider to your layout and then add the following code to the `slider1 Callback` callback:

   ```
   handles.activex1.radius = ...
       get(hObject,'Value')*handles.default_radius;
   handles.activex1.label = ...
       ['Radius = ' num2str(handles.activex1.radius)];
   refresh(handles.figure1);
   ```

   The first command

   - Gets the `Value` of the slider, which in this example is a number between 0 and 1, the default values of the slider's `Min` and `Max` properties.
   - Sets `handles.activex1.radius` equal to the `Value` times the default radius.

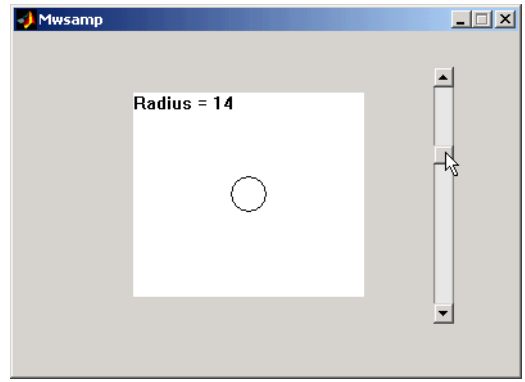
2. In the opening function, add the default radius to the handles structure. The `activex1_Click` callback uses the default radius to update the slider value if the user clicks the circle.

```
handles.default_radius = handles.activex1.radius;
```

3 In the `activex1_Click` callback, reset the slider's `Value` each time the user clicks the circle in the ActiveX control. The following command causes the slider to change position corresponding to the new value of the radius.

```
set(handles.slider1,'Value',...
    hObject.radius/handles.default_radius);
```

When you open the GUI and move the slider by clicking and dragging, the radius changes to a new value between 0 and the default radius of 20, as shown in the following figure.



Viewing the Methods for an ActiveX Control

To view the available methods for an ActiveX control, you first need to obtain the handle to the control. One way to do this is the following:

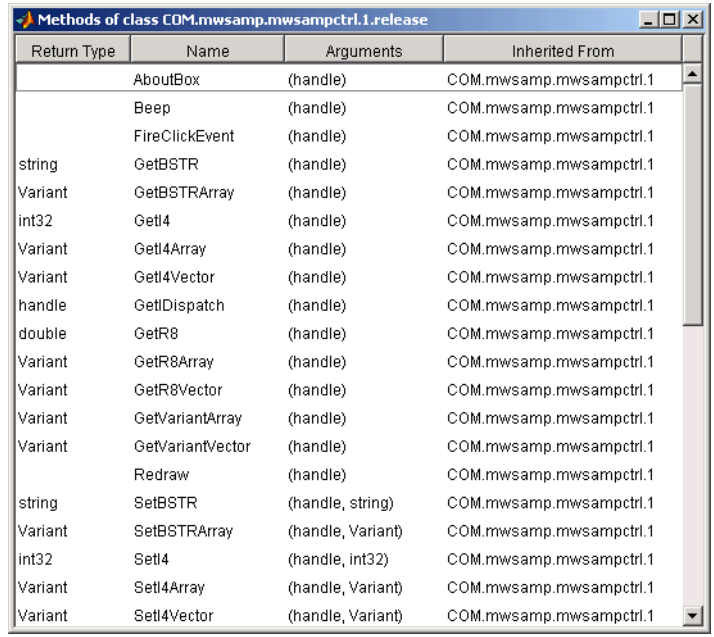
1 In the GUI M-file, add the command `keyboard` on a separate line of the `activex1_Click` callback. The command `keyboard` puts MATLAB in debug mode and pauses at the `activex1_Click` callback when you click the ActiveX control.

2 Run the GUI and click the ActiveX control.

The handle to the control is now set to `hObject`. To view the methods for the control, enter

```
methodsview(hObject)
```

at the MATLAB prompt. This displays the available methods in a new window, as shown in the following figure.



Return Type	Name	Arguments	Inherited From
	AboutBox	(handle)	COM.mwsamp.mwsampctrl.1
	Beep	(handle)	COM.mwsamp.mwsampctrl.1
	FireClickEvent	(handle)	COM.mwsamp.mwsampctrl.1
string	GetBSTR	(handle)	COM.mwsamp.mwsampctrl.1
Variant	GetBSTRArray	(handle)	COM.mwsamp.mwsampctrl.1
int32	GetI4	(handle)	COM.mwsamp.mwsampctrl.1
Variant	GetI4Array	(handle)	COM.mwsamp.mwsampctrl.1
Variant	GetI4Vector	(handle)	COM.mwsamp.mwsampctrl.1
handle	GetIDispatch	(handle)	COM.mwsamp.mwsampctrl.1
double	GetR8	(handle)	COM.mwsamp.mwsampctrl.1
Variant	GetR8Array	(handle)	COM.mwsamp.mwsampctrl.1
Variant	GetR8Vector	(handle)	COM.mwsamp.mwsampctrl.1
Variant	GetVariantArray	(handle)	COM.mwsamp.mwsampctrl.1
Variant	GetVariantVector	(handle)	COM.mwsamp.mwsampctrl.1
	Redraw	(handle)	COM.mwsamp.mwsampctrl.1
string	SetBSTR	(handle, string)	COM.mwsamp.mwsampctrl.1
Variant	SetBSTRArray	(handle, Variant)	COM.mwsamp.mwsampctrl.1
int32	SetI4	(handle, int32)	COM.mwsamp.mwsampctrl.1
Variant	SetI4Array	(handle, Variant)	COM.mwsamp.mwsampctrl.1
Variant	SetI4Vector	(handle, Variant)	COM.mwsamp.mwsampctrl.1

Alternatively, you can enter

```
methods(hObject)
```

which displays the available methods in the Command Window.

For more information about methods for ActiveX controls, see Invoking Methods in the online MATLAB documentation. See the reference pages for `methodsview` and `methods` for more information about these functions.

Saving a GUI that Contains an ActiveX Control

When you save a GUI that contains ActiveX controls, GUIDE creates a file in the current directory for each such control. The filename consists of the name of the GUI followed by an underscore (_) and `activex`*n*, where `n` is a sequence number. For example, if the GUI is named `mygui` then the filename would be `mygui_activex1`. The filename does not have an extension.

Compiling a GUI that Contains an ActiveX Control

If you use the MATLAB Compiler `mcc` command to compile a GUI that contains an ActiveX control, you must use the `-a` flag to add the ActiveX file, which GUIDE saves in the current directory, to the CTF archive. Your command should be similar to this

```
mcc -m mygui -a mygui_activex1
```

where `mygui_activex1` is the name of the ActiveX file. See the MATLAB Compiler documentation for more information. If you have more than one such file, use a separate `-a` flag for each file. You must have installed the MATLAB Compiler to compile a GUI.

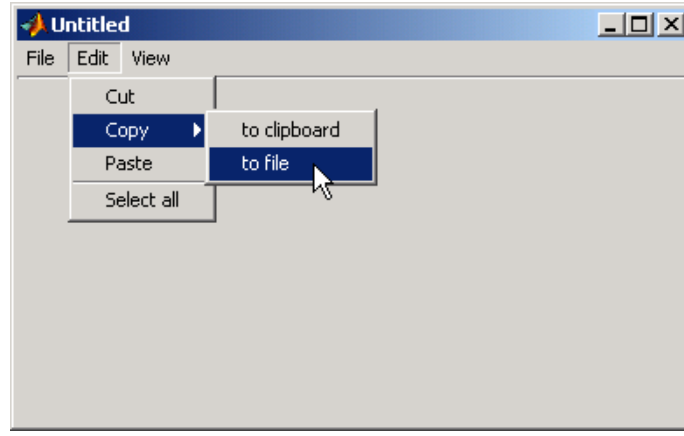
Menu Item

The Menu Editor generates an empty callback subfunction for every menu item, including menu titles.

Programming a Menu Title

Because clicking on a menu title automatically displays the menu below it, you may not need to program callbacks at the title level. However, the callback associated with a menu title can be a good place to enable or disable menu items below it.

Consider the example illustrated in the following picture.



When a user selects the **to file** option under the **Edit** menu's **Copy** option, only the **to file** callback is required to perform the action.

Suppose, however, that only certain objects can be copied to a file. You can use the **Copy** item `Callback` callback to enable or disable the **to file** item, depending on the type of object selected.

Opening a Dialog Box from a Menu Callback

The `Callback` callback for the **to file** menu item could contain code such as the following to display the standard dialog box for saving files.

```
[file,path] = uiputfile('animinit.m','Save file name');
```

`'Save file name'` is the dialog box title. In the dialog box, the filename field is set to `animinit.m`, and the filter set to M-files (`*.m`). For more information, see the `uiputfile` reference page.

Updating a Menu Item Check

A check is useful to indicate the current state of some menu items. If you selected **Check mark this item** in the Menu Editor, the item initially appears checked. Each time the user selects the menu item, the callback for that item must turn the check `on` or `off`. The following example shows you how to do this by changing the value of the menu item's `Checked` property.

```
if strcmp(get(gcbo, 'Checked'),'on')
    set(gcbo,'Checked','off');
else
    set(gcbo,'Checked','on');
end
```

The `gcbo` function returns the handle of the object whose callback is executing, in this case the menu item. The `strcmp` function compares two strings and returns logical `1` (true) if the two are identical and logical `0` (false) otherwise.

If you set the axes to be visible when a user first opens the GUI, make sure to select the **Check mark this item** check box in the Menu Editor, so that a check will appear next to the **Show axes** menu item initially.

Note From the Menu Editor, you can view a menu item's `Callback` callback in your editor by selecting the menu item and clicking the **View** button.

9

Managing Application-Defined Data

Mechanisms for Managing Data (p. 9-2)	Describes various mechanisms for managing application-defined data. Explains how GUIDE uses one of these mechanisms, GUI data.
Sharing Data Among a GUI's Callbacks (p. 9-7)	Shows how each mechanism for managing data can be used to share data among a GUI's callbacks.

Mechanisms for Managing Data

Most GUIs generate or use data that is specific to the application. This topic describes the three mechanisms for managing application-defined data in the GUI environment. They provide a way for applications to save and retrieve data stored with the GUI.

- "GUI Data" on page 9-2
- "Application Data" on page 9-4
- "UserData Property" on page 9-6

The GUI data and application data mechanisms are similar but GUI data can be simpler to use. GUIDE specifically uses GUI data to manage the `handles` structure, but you can either use the GUI data `handles` structure or application data to manage application-defined data. The `UserData` property can also hold application-defined data.

GUI Data

GUI data is managed using the `guidata` function. This function can store a single variable as GUI data. It is also used to retrieve the value of that variable.

- "About GUI Data" on page 9-2
- "GUI Data in GUIDE" on page 9-3
- "Adding Fields to the handles Structure" on page 9-3
- "Changing GUI Data in an M-File Generated by GUIDE" on page 9-4

About GUI Data

GUI data is always associated with the GUI figure. It is available to all callbacks of all components of the GUI. If you specify a component handle when you save or retrieve GUI data, MATLAB automatically associates the data with the component's parent figure.

GUI data can contain only one variable at any time. Writing GUI data overwrites the existing GUI data. For this reason, GUI data is usually defined to be a structure to which you can add fields as you need them.

GUI data provides application developers with a convenient interface to a figure's application data:

- You do not need to create and maintain a hard-coded name for the data throughout your source code.
- You can access the data from within a callback routine using the component's handle, without having to find the figure handle. For GUIDE users, the object handle is automatically passed to each callback as `hObject`.

GUI Data in GUIDE

GUIDE automatically uses `guidata` to create and maintain the `handles` structure. This structure is automatically passed as an argument to every callback. The `handles` structure contains the handles of all components in the GUI.

In a GUI created using GUIDE, you cannot use `guidata` to manage any variable other than the `handles` structure. If you do, you may overwrite the `handles` structure and your GUI will not work. If you want to use GUI data to share application-defined data among callbacks, you must save the data in fields that you add to the `handles` structure.

Note that the GUIDE templates use the `handles` structure to store application-defined data. See "Selecting a GUI Template" on page 6-5 for information about the templates.

Note For more information, see "The Handles Structure" on page 8-13.

Adding Fields to the handles Structure

To add a field to the `handles` structure, which is passed as an argument to every callback in GUIDE.

1. Assign a value to the new field. This adds the field to the structure. For example

   ```
   handles.number_errors = 0;
   ```

   adds the field `number_errors` to the structure `handles` and sets it to `0`.

2 Use the following command to save the data.

```
guidata(hObject,handles)
```

where `hObject` is the handle of the component for which the callback was triggered. It is passed automatically to every callback.

Changing GUI Data in an M-File Generated by GUIDE

In a GUIDE-generated M-file, GUI data is always represented by the `handles` structure. This example updates the `handles` structure and then saves it.

1 Assume that the `handles` structure contains an application-defined field `handles.when` whose value is `'now'`.

2 In a GUI callback, make the desired change to the `handles` structure. This example changes the value of `handles.when` to `'later'`, but does not save the `handles` structure.

```
handles.when = 'later';
```

3 Save the changed version of the `handles` structure with the command

```
guidata(hObject,handles)
```

where `hObject`, which is passed automatically to every callback, is the handle of the component for which the callback was triggered. If you do not save the `handles` structure, the change you made to it in the preceding step is lost.

Application Data

Application data provides a way for applications to save and retrieve data associated with a specified object. For a GUI, this is usually the GUI figure but can also be any component. The data is stored as name/value pairs. Application data enables you to create what are essentially user-defined properties for an object.

The following table summarizes the functions that provide access to application data. For more detailed information, see the individual function reference pages.

Functions for Managing Application Data

Function	Purpose
`setappdata`	Specify named application data for an object. The object does not have to be a figure. You can specify more than one named application data for an object. However, each name must be unique for that object and can be associated with only one value, usually a structure.
`getappdata`	Retrieve named application data. To retrieve named application data, you must know the name associated with the application data and the handle of the object with which it is associated.
`isappdata`	True if the named application data exists.
`rmappdata`	Remove the named application data.

Creating Application Data in GUIDE

Use the `setappdata` function to create application data. This example generates a 35-by-35 matrix of normally distributed random numbers in the opening function and creates application data `mydata` to manage it.

```
function mygui_OpeningFcn(hObject, eventdata, handles, varargin)
matrices.rand_35 = randn(35);
setappdata(hObject,'mydata',matrices);
```

Because this code appears in the opening function, `hObject` is the handle of the GUI figure, and the code associates `mydata` with the figure.

Adding Fields to an Application Data Structure in GUIDE

Application data is usually defined as a structure to enable you to add fields as necessary. In this example, a push button adds a field to the application data structure `mydata` created in the previous topic.

1. Use `getappdata` to retrieve the structure.

   From the example in the previous topic, the name of the application data structure is `mydata`. It is associated with the figure whose `Tag` is `figure1`. Since the `handles` structure is passed to every callback, the code can specify the figure's handle as `handles.figure1`.

   ```
   function mygui_pushbutton1(hObject, eventdata, handles)
   matrices = getappdata(handles.figure1,'mydata');
   ```

2. Create a new field and assign it a value. For example

   ```
   matrices.randn_50 = randn(50);
   ```

   adds the field `randn_50` to the `matrices` structure and sets it to a 50-by-50 matrix of normally distributed random numbers.

3. Use `setappdata` to save the data. This example uses `setappdata` to save the `matrices` structure as the application data structure `mydata`.

   ```
   setappdata(hObject,'mydata',matrices);
   ```

UserData Property

All GUI components, including menus, and the figure have a `UserData` property. You can assign any valid MATLAB value to the `UserData` property. To retrieve the data, a callback must know the handle of the component with which the data is associated.

1. In this example, an edit text stores the user-entered string in its `UserData` property.

   ```
   function mygui_edittext1(hObject, eventdata, handles)
   mystring = get(hObject,'String');
   set(hObject,'UserData',mystring);
   ```

2. A push button retrieves the string from the edit text `UserData` property. The callback uses the `handles` structure and the edit text `Tag` property, `edittext1`, to specify the edit text handle.

   ```
   function mygui_pushbutton1(hObject, eventdata, handles)
   string = get(handles.edittext1,'UserData');
   ```

Sharing Data Among a GUI's Callbacks

You can use GUI data, application data, and the UserData property to share data among a GUI's callbacks. This topic shows you how for each of these data forms:

- "GUI Data" on page 9-7
- "Application Data" on page 9-9
- "UserData Property" on page 9-11

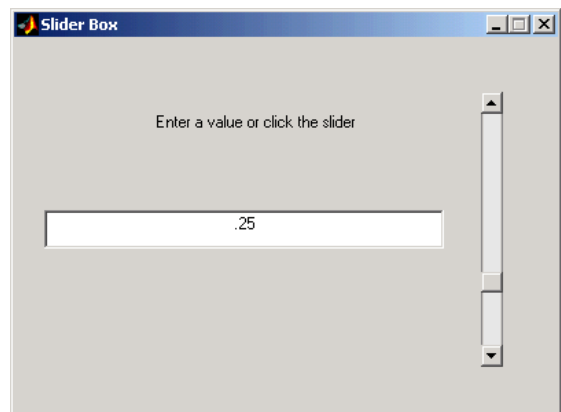
See "Mechanisms for Managing Data" on page 9-2 for general information about these data forms.

GUI Data

GUI data, which is managed by the `guidata` function, is accessible to all callbacks of the GUI. A callback for one component can set a value in GUI data which can then be read by a callback for another component. See "GUI Data" on page 9-2 for more information about GUI data.

GUI Data Example: Passing Data Between a Slider and an Edit Text Box

This example uses a GUI that contains a slider and an edit text box as shown in the following figure. A static text instructs the user to enter a value or click the slider. The example uses GUI data to initialize and maintain an error counter.



The GUI behavior is as follows:

- When a user moves the slider, the text box displays the slider's current value.
- When a user types a value into the text box, the slider updates to this value.
- If a user enters a value in the text box that is out of range for the slider — that is, a value that is not between 0 an 1 — the application returns a message in the edit text box indicating how many times the user has entered an erroneous value.

The commands given in the following steps initialize the error counter and implement the interchange between the slider and the edit text box.

1. Define the error counter in the opening function. The GUI records the number of times a user enters an erroneous value in the text box and stores this number in a field of the `handles` structure. Start by defining this field, called `number_errors`, in the opening function as follows:

   ```
   handles.number_errors = 0;
   ```

   Type this statement before the following line, which GUIDE automatically inserts into the opening function.

   ```
   guidata(hObject,handles); % Save the updated handles structure
   ```

   The `guidata` command saves the `handles` structure so that it can be retrieved in the GUI's callbacks.

2. Set the edit text value from the slider `Callback` callback. The following command in the slider `Callback` updates the value displayed in the edit text when a user moves the slider and releases the mouse button.

   ```
   set(handles.edit1,'String',...
      num2str(get(handles.slider1,'Value')));
   ```

   The code combines three commands:

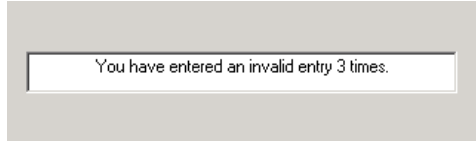
   - The `get` command obtains the current value of the slider.
   - The `num2str` command converts the value to a string.
   - The `set` command sets the `String` property of the edit text to the updated value.

3 Set the slider value from the edit text `Callback` callback. The edit text `Callback` sets the slider's value to the number the user types in, after checking to see if it is a single numeric value between 0 and 1. If the value is out of range, then the error count is incremented and the edit text displays a message telling the user how many times they have entered an invalid number.

```
val = str2double(get(handles.edit1,'String'));
% Determine whether val is a number between 0 and 1
if isnumeric(val) & length(val)==1 & ...
   val >= get(handles.slider1,'Min') & ...
   val <= get(handles.slider1,'Max')
   set(handles.slider1,'Value',val);
else
% Increment the error count, and display it.
   handles.number_errors = handles.number_errors+1;
   guidata(hObject,handles); % store the changes
   set(handles.edit1,'String',...
   ['You have entered an invalid entry ',...
num2str(handles.number_errors),' times.']);
end
```

If the user types a number between 0 and 1 in the edit box and then presses **Enter** or clicks outside the edit box, the `Callback` sets `handles.slider1` to the new value and the slider moves to the corresponding position.

If the entry is invalid — for example, `2.5` — the GUI increments the value of `handles.number_errors` and displays a message like the following in the edit text box:



Application Data

Application data can be associated with any object – a component, menu, or the figure itself. To access application data, a callback must know the name of the data and the handle of the component with which it is associated. Use the

functions `setappdata`, `getappdata`, `isappdata`, and `rmappdata` to manage application data.

See "Application Data" on page 9-4 for more information about application data.

Application Data Example: Passing Data Between a Slider and an Edit Text Box

The previous topic, "GUI Data Example: Passing Data Between a Slider and an Edit Text Box" on page 9-7, used GUI data to initialize and maintain an error counter. This example shows you how to do the same thing using application data. Refer to the previous topic for details of the example.

1. Define the error counter in the opening function. Add the following code to the opening function. This code first creates a structure `slider_data`, then assigns it to the named application data `slider`. Because this code appears in the opening function, using `hObject` associates the application data with the figure.

   ```
   slider_data.number_errors = 0;
   setappdata(hObject,'slider',slider_data);
   ```

2. Set the edit text value from the slider `Callback` callback. Add this statement to the slider `Callback`.

   ```
   set(handles.edit1,'String',...
      num2str(get(handles.slider1,'Value')));
   ```

3. Set the slider value from the edit text `Callback` callback. Add this code to the edit text `Callback`. It assumes the figure's `Tag` property is `figure1`.

   To update the number or errors, the code must first retrieve the named application data `slider`, then increment the count. It then saves the application data and displays the new error count.

   ```
   val = str2double(get(hObject,'String'));
   % Determine whether val is a number between 0 and 1
   if isnumeric(val) & length(val)==1 & ...
      val >= get(handles.slider1,'Min') & ...
      val <= get(handles.slider1,'Max')
      set(handles.slider1,'Value',val);
   else
   ```

```
% Retrieve and increment the error count.
   slider_data = getappdata(handles.figure1,'slider');
   slider_data.number_errors = slider_data.number_errors+1;
% Save the changes.
   setappdata(handles.figure1,'slider',slider_data);
% Display new total.
   set(hObject,'String',...
   ['You have entered an invalid entry ',...
num2str(slider_data.number_errors),' times.']);
end
```

UserData Property

Every GUI component, and the figure itself, has a `UserData` property that you can use to store application defined data. To access `UserData`, a callback must know the handle of the component with which it is associated.

Use the `get` function to retrieve `UserData`, and the `set` function to set it.

UserData Property Example: Passing Data Between a Slider and an Edit Text Box

A previous topic, "GUI Data Example: Passing Data Between a Slider and an Edit Text Box" on page 9-7, used GUI data to initialize and maintain an error counter. This example shows you how to do the same thing using the `UserData` property of the edit text box to store the error count. Refer to the GUI data example for details of the example.

1 Initialize the edit text `UserData` property in the opening function. Add the following code to the opening function. This code initializes the data in a structure to allow for other data that may be needed.

```
number_errors = 0;
set(handles.edit1,'UserData',number_errors)
```

Note Alternatively, you could add a `CreateFcn` callback for the edit text to the M-file, and initialize the error counter there.

---

2. Set the edit text value from the slider `Callback` callback. Add this statement to the slider `Callback`.

```
set(handles.edit1,'String',...
   num2str(get(handles.slider1,'Value')));
```

3. Set the slider value from the edit text `Callback` callback. To do this, add the following code to the edit text `Callback`.

   To update the number or errors, the code must first retrieve the value of the edit text `UserData` property, then increment the count. It then saves the updated error count in the `UserData` property and displays the new count.

```
val = str2double(get(hObject,'String'));
% Determine whether val is a number between 0 and 1
if isnumeric(val) & length(val)==1 & ...
   val >= get(handles.slider1,'Min') & ...
   val <= get(handles.slider1,'Max')
   set(handles.slider1,'Value',val);
else
% Retrieve and increment the error count.
   number_errors = get(hObject,'UserData');
   number_errors = number_errors+1;
% Save the changes.
   set(hObject,'UserData',slider_data);
% Display new total.
   set(hObject,'String',...
   ['You have entered an invalid entry ',...
num2str(number_errors),' times.']);
end
```

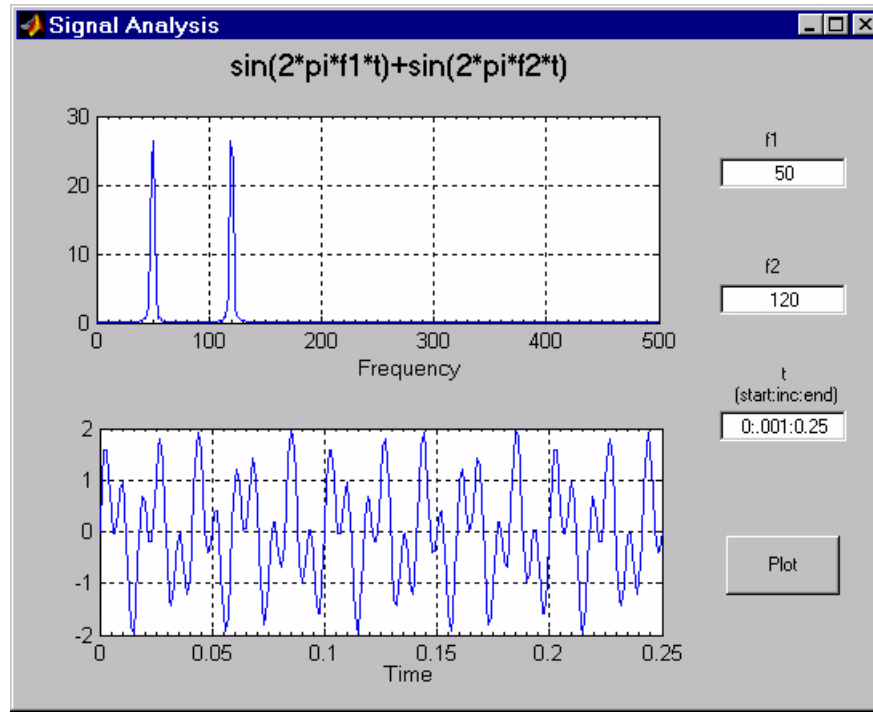
10

Examples of GUIDE GUIs

GUI with Multiple Axes (p. 10-2)	Analyze data and generate frequency and time domain plots in the GUI figure.
List Box Directory Reader (p. 10-9)	List the contents of a directory, navigate to other directories, and define what command to execute when users double-click on a given type of file.
Accessing Workspace Variables from a List Box (p. 10-16)	List variables in the base MATLAB workspace from a GUI and plot them. This example illustrates selecting multiple items and executing commands in a different workspace.
A GUI to Set Simulink Model Parameters (p. 10-20)	Set parameters in a Simulink® model, save and plot the data, and implement a help button.
An Address Book Reader (p. 10-32)	Read data from MAT-files, edit and save the data, and manage GUI data using the `handles` structure.
Using a Modal Dialog to Confirm an Operation (p. 10-47)	Illustrates use of a modal dialog GUI to confirm that the user wants to proceed with an operation.

GUI with Multiple Axes

This example creates a GUI that contains two axes for plotting data. For simplicity, this example obtains data by evaluating an expression using parameters entered by the user.



Techniques Used in the Example

GUI-building techniques illustrated in this example include

- Controlling which axes is the target for plotting commands.
- Using edit text controls to read numeric input and MATLAB expressions.

View Completed Layout and Its GUI M-File

If you are reading this in the MATLAB Help browser, you can click the following links to display the GUIDE Layout Editor and the MATLAB Editor with a completed version of this example. This enables you to see the values of all component properties and to understand how the components are assembled to create the GUI. You can also see a complete listing of the code that is discussed in the following sections.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

- Click here to display this GUI in the Layout Editor.
- Click here to display the GUI M-file in the MATLAB Editor.

Design of the GUI

This GUI requires three input values:

- Frequency one (`f1`)
- Frequency two (`f2`)
- A time vector (`t`)

When the user clicks the **Plot** button, the GUI puts these values into a MATLAB expression that is the sum of two sine function:

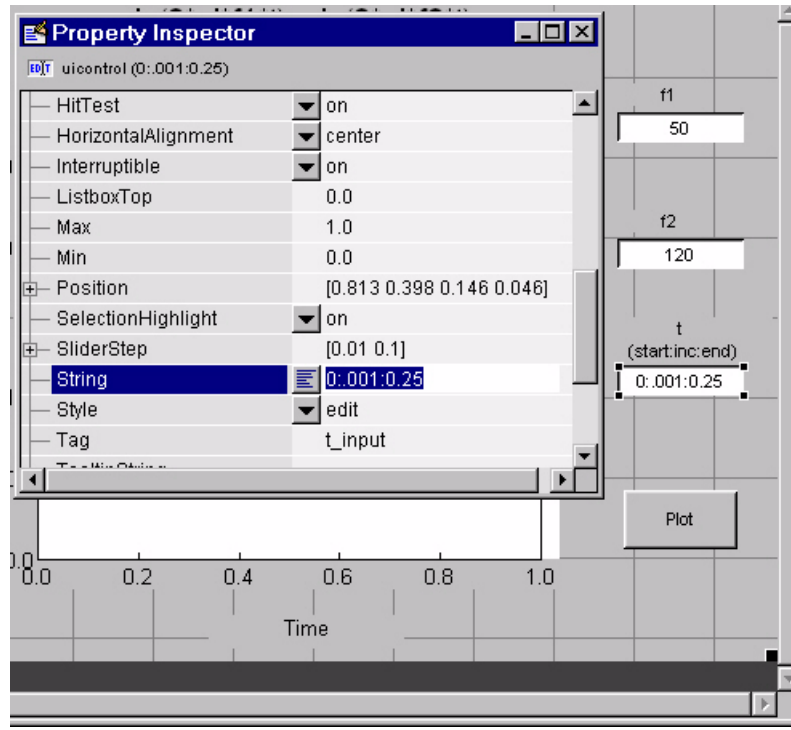
```
x = sin(2*pi*f1*t) + sin(2*pi*f2*t)
```

The GUI then calculates the FFT of `x` and creates two plots — one frequency domain and one time domain.

Specifying Default Values for the Inputs

The GUI uses default values for the three inputs. This enables users to click on the **Plot** button and see a result as soon as the GUI is run. It also helps to indicate what values the user might enter.

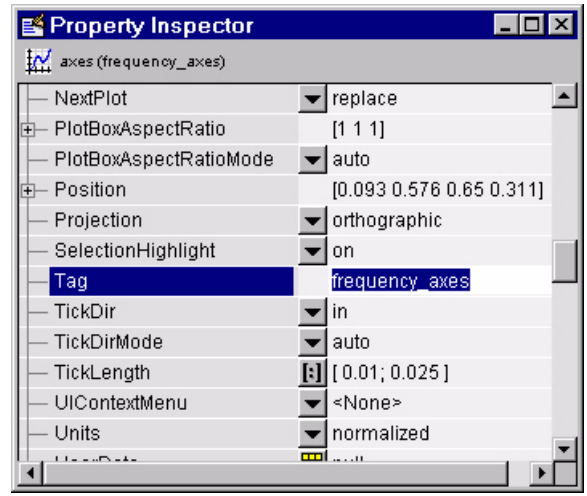
To create the default values, set the `String` property of the edit text. The following figure shows the value set for the time vector.



Identifying the Axes

Since there are two axes in this GUI, you must be able to specify which one you want to target when you issue the plotting commands. To do this, use the `handles` structure, which contains the handles of all components in the GUI.

The field name in the `handles` structure that contains the handle of any given component is derived from the component's `Tag` property. To make code more readable (and to make it easier to remember) this example sets the `Tag` property to descriptive names.



For example, the `Tag` of the axes used to display the FFT is set to `frequency_axes`. Therefore, within a callback, you access its handle with

```
handles.frequency_axes
```

Likewise, the `Tag` of the time axes is set to `time_axes`.

See “The Handles Structure” on page 8-13 for more information on the `handles` structure. See “Plot Push Button Callback” on page 10-6 for the details of how to use the handle to specify the target axes.

GUI Option Settings

There are two GUI option settings that are particularly important for this GUI:

- Resize behavior: **Proportional**
- Command-line accessibility: **Callback**

Proportional Resize Behavior. Selecting **Proportional** as the resize behavior enables users to change the GUI to better view the plots. The components change size in proportion to the GUI figure size. This generally produces good results except when extremes of dimensions are used.

Callback Accessibility of Object Handles. When GUIs include axes, handles should be visible from within callbacks. This enables you to use plotting commands

like you would on the command line. Note that **Callback** is the default setting for command-line accessibility.

See “GUI Options” on page 5-9 for more information.

Plot Push Button Callback

This GUI uses only the **Plot** button callback; the edit text callbacks are not needed and have been deleted from the GUI M-file. When a user clicks the **Plot** button, the callback performs three basic tasks — it gets user input from the edit text components, calculates data, and creates the two plots.

Getting User Input

The three edit text boxes provide a way for the user to enter values for the two frequencies and the time vector. The first task for the callback is to read these values. This involves:

- Reading the current values in the three edit text boxes using the `handles` structure to access the edit text handles.
- Converting the two frequency values (`f1` and `f2`) from string to doubles using `str2double`.
- Evaluating the time string using `eval` to produce a vector `t`, which the callback used to evaluate the mathematical expression.

The following code shows how the callback obtains the input.

```
% Get user input from GUI
f1 = str2double(get(handles.f1_input,'String'));
f2 = str2double(get(handles.f2_input,'String'));
t = eval(get(handles.t_input,'String'));
```

Calculating Data

Once the input data has been converted to numeric form and assigned to local variables, the next step is to calculate the data needed for the plots. See the `fft` function for an explanation of how this is done.

Targeting Specific Axes

The final task for the callback is to actually generate the plots. This involves

- Making the appropriate axes current using the `axes` command and the handle of the axes. For example,

  ```
  axes(handles.frequency_axes)
  ```

- Issuing the `plot` command.
- Setting any properties that are automatically reset by the `plot` command.

The last step is necessary because many plotting commands (including `plot`) clear the axes before creating the graph. This means you cannot use the Property Inspector to set the `XMinorTick` and grid properties that are used in this example, since they are reset when the callback executes `plot`.

When looking at the following code listing, note how the `handles` structure is used to access the handle of the axes when needed.

Plot Button Code Listing

```
function plot_button_Callback(hObject, eventdata, handles)
% hObject    handle to plot_button (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get user input from GUI
f1 = str2double(get(handles.f1_input,'String'));
f2 = str2double(get(handles.f2_input,'String'));
t = eval(get(handles.t_input,'String'));

% Calculate data
x = sin(2*pi*f1*t) + sin(2*pi*f2*t);
y = fft(x,512);
m = y.*conj(y)/512;
f = 1000*(0:256)/512;

% Create frequency plot
axes(handles.frequency_axes) % Select the proper axes
plot(f,m(1:257))
set(handles.frequency_axes,'XMinorTick','on')
grid on

% Create time plot
axes(handles.time_axes) % Select the proper axes
```

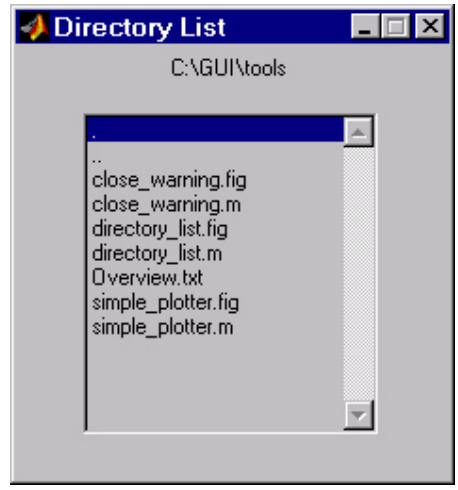
```
plot(t,x)
set(handles.time_axes,'XMinorTick','on')
grid on
```

List Box Directory Reader

This example uses a list box to display the files in a directory. When the user double clicks on a list item, one of the following happens:

- If the item is a file, the GUI opens the file appropriately for the file type.
- If the item is a directory, the GUI reads the contents of that directory into the list box.
- If the item is a single dot (.), the GUI updates the display of the current directory.
- If the item is two dots (..), the GUI changes to the directory up one level and populates the list box with the contents of that directory.

The following figure illustrates the GUI.



View Layout and GUI M-File

If you are reading this in the MATLAB Help browser, you can click the following links to display the GUIDE Layout Editor and the MATLAB Editor with a completed version of this example. This enables you to see the values of all component properties and to understand how the components are assembled to create the GUI. You can also see a complete listing of the code that is discussed in the following sections.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

- Click here to display this GUI in the Layout Editor.
- Click here to display the GUI M-file in the editor.

Implementing the GUI

The following sections describe the implementation:

- "Specifying the Directory to List" — shows how to pass a directory path as input argument when the GUI is run.
- "Loading the List Box" — describes the subfunction that loads the contents of the directory into the list box. This subfunction also saves information about the contents of a directory in the `handles` structure.
- "The List Box Callback" — explains how the list box is programmed to respond to user double clicks on items in the list box.

Specifying the Directory to List

You can specify the directory to list when the GUI is first opened by passing the string `'create'` and a string containing the full path to the directory as arguments. The syntax for doing this is `list_box('create','dir_path')`. If you do not specify a directory (i.e., if you call the GUI M-file with no input arguments), the GUI then uses the MATLAB current directory.

The default behavior of the GUI M-file that GUIDE generates is to run the GUI when there are no input arguments or to call a subfunction when the first input argument is a character string. This example changes this behavior so that you can call the M-file with

- No input arguments — run the GUI using the MATLAB current directory.
- First input argument is `'dir'` and second input argument is a string that specifies a valid path to a directory — run the GUI, displaying the specified directory.

- First input argument is not a directory, but is a character string and there is more than one argument — execute the subfunction identified by the argument (execute callback).

The following code listing show the setup section of the GUI M-file, which does one the following:

- Sets the list box directory to the current directory, if no directory is specified.
- Changes the current directory, if a directory is specified.

```
function lbox2_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to untitled (see VARARGIN)

% Choose default command line output for lbox2
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

if nargin == 3,
    initial_dir = pwd;
elseif nargin > 4
    if strcmpi(varargin{1},'dir')
        if exist(varargin{2},'dir')
            initial_dir = varargin{2};
        else
            errordlg({'Input argument must be a valid',...
                      'directory'},'Input Argument Error!')
            return
        end
    else
        errordlg('Unrecognized input argument',...
                 'Input Argument Error!');
        return;
    end
end
```

```
% Populate the listbox
load_listbox(initial_dir,handles)
```

Loading the List Box

This example creates a subfunction to load items into the list box. This subfunction accepts the path to a directory and the `handles` structure as input arguments. It performs these steps:

- Change to the specified directory so the GUI can navigate up and down the tree as required.
- Use the `dir` command to get a list of files in the specified directory and to determine which name is a directory and which is a file. `dir` returns a structure (`dir_struct`) with two fields, `name` and `isdir`, which contain this information.
- Sort the file and directory names (`sortrows`) and save the sorted names and other information in the `handles` structure so this information can be passed to other functions.

  The `name` structure field is passed to `sortrows` as a cell array, which is transposed to get one file name per row. The `isdir` field and the sorted index values, `sorted_index`, are saved as vectors in the `handles` structure.
- Call `guidata` to save the `handles` structure.
- Set the list box `String` property to display the file and directory names and set the `Value` property to `1`. This is necessary to ensure `Value` never exceeds the number of items in `String`, since MATLAB updates the `Value` property only when a selection occurs and not when the contents of `String` changes.
- Displays the current directory in the text box by setting its `String` property to the output of the `pwd` command.

The `load_listbox` function is called by the opening function of the GUI M-file as well as by the list box callback.

```
function load_listbox(dir_path, handles)
cd (dir_path)
dir_struct = dir(dir_path);
[sorted_names,sorted_index] = sortrows({dir_struct.name}');
handles.file_names = sorted_names;
handles.is_dir = [dir_struct.isdir];
handles.sorted_index = sorted_index;
```

```
guidata(handles.figure1,handles)
set(handles.listbox1,'String',handles.file_names,...
   'Value',1)
set(handles.text1,'String',pwd)
```

The List Box Callback

The list box callback handles only one case: a double-click on an item. Double clicking is the standard way to open a file from a list box. If the selected item is a file, it is passed to the `open` command; if it is a directory, the GUI changes to that directory and lists its contents.

Defining How to Open File Types

The callback makes use of the fact that the `open` command can handle a number of different file types. However, the callback treats FIG-files differently. Instead of opening the FIG-file, it passes it to the `guide` command for editing.

Determining Which Item the User Selected

Since a single click on an item also invokes the list box callback, it is necessary to query the figure `SelectionType` property to determine when the user has performed a double click. A double-click on an item sets the `SelectionType` property to `open`.

All the items in the list box are referenced by an index from `1` to `n`, where `1` refers to the first item and `n` is the index of the `n`th item. MATLAB saves this index in the list box `Value` property.

The callback uses this index to get the name of the selected item from the list of items contained in the `String` property.

Determining if the Selected Item is a File or Directory

The `load_listbox` function uses the `dir` command to obtain a list of values that indicate whether an item is a file or directory. These values (1 for directory, 0 for file) are saved in the `handles` structure. The list box callback queries these values to determine if current selection is a file or directory and takes the following action:

- If the selection is a directory — change to the directory (`cd`) and call `load_listbox` again to populate the list box with the contents of the new directory.

- If the selection is a file — get the file extension (`fileparts`) to determine if it is a FIG-file, which is opened with `guide`. All other file types are passed to `open`.

The `open` statement is called within a `try/catch` block to capture errors in an error dialog (`errordlg`), instead of returning to the command line.

```
get(handles.figure1,'SelectionType');
% If double click
if strcmp(get(handles.figure1,'SelectionType'),'open')
    index_selected = get(handles.listbox1,'Value');
    file_list = get(handles.listbox1,'String');
    % Item selected in list box
    filename = file_list{index_selected};
    % If directory
    if  handles.is_dir(handles.sorted_index(index_selected))
        cd (filename)
        % Load list box with new directory.
        load_listbox(pwd,handles)
    else
        [path,name,ext,ver] = fileparts(filename);
        switch ext
            case '.fig'
                % Open FIG-file with guide command.
                guide (filename)
            otherwise
                try
                    % Use open for other file types.
                    open(filename)
                catch
                    errordlg(lasterr,'File Type Error','modal')
                end
        end
    end
end
```

Opening Unknown File Types

You can extend the file types that the `open` command recognizes to include any file having a three-character extension. You do this by creating an M-file with the name `openxyz`, where `xyz` is the extension. Note that the list box callback

does not take this approach for FIG-files since `openfig.m` is required by the GUI M-file. See `open` for more information.

Accessing Workspace Variables from a List Box

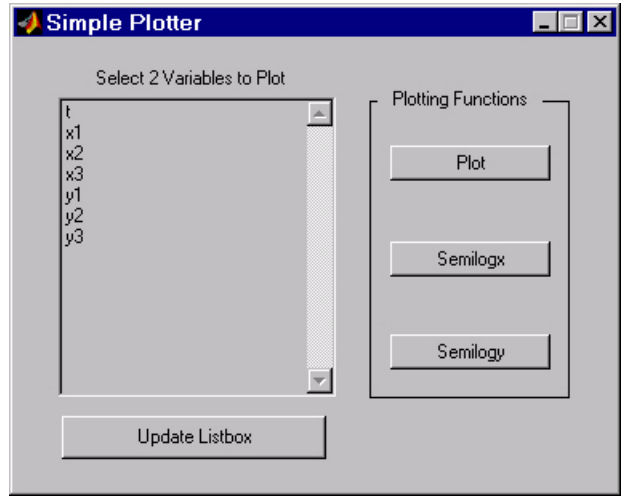
This GUI uses a list box to display workspace variables, which the user can then plot.

Techniques Used in This Example

This example demonstrates how to:

- Populate the list box with the variable names that exist in the base workspace.
- Display the list box with no items initially selected.
- Enable multiple item selection in the list box.
- Update the list items when the user press a button.
- Evaluate the plotting commands in the base workspace.

The following figure illustrates the layout.



Note that the list box callback is not used in this program because the plotting actions are initiated by push buttons. In this situation you must do one of the following:

- Leave the empty list box callback in the GUI M-file.
- Delete the string assigned to the list box `Callback` property.

View Completed Layout and Its GUI M-File

If you are reading this in the MATLAB Help browser, you can click the following links to display the GUIDE Layout Editor and the MATLAB Editor with a completed version of this example. This enables you to see the values of all component properties and to understand how the components are assembled to create the GUI. You can also see a complete listing of the code that is discussed in the following sections.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

- Click here to display this GUI in the Layout Editor.
- Click here to display the GUI M-file in the editor.

Reading Workspace Variables

When the GUI initializes, it needs to query the workspace variables and set the list box `String` property to display these variable names. Adding the following subfunction to the GUI M-file accomplishes this using `evalin` to execute the `who` command in the base workspace. The `who` command returns a cell array of strings, which are used to populate the list box.

```
function update_listbox(handles)
vars = evalin('base','who');
set(handles.listbox1,'String',vars)
```

The function's input argument is the `handles` structure generated by the GUI M-file. This structure contains the handle of the list box, as well as the handles all other components in the GUI.

The callback for the **Update Listbox** push button also calls `update_listbox`.

Reading the Selections from the List Box

This GUI requires the user to select two variables from the workspace and then choose one of three plot commands to create the graph: `plot`, `semilogx`, or `semilogy`.

Enabling Multiple Selection

To enable multiple selection in a list box, you must set the `Min` and `Max` properties so that `Max  Min > 1`. This requires you to change the default `Min` and `Max` values of `0` and `1` to meet these conditions. Use the Property Inspector to set these properties on the list box.

How Users Select Multiple Items

List box multiple selection follows the standard for most systems:

- **Control**-click left mouse button — noncontiguous multi-item selection
- **Shift**-click left mouse button — contiguous multi-item selection

Users must use one of these techniques to select the two variables required to create the plot.

Returning Variable Names for the Plotting Functions

The `get_var_names` subroutine returns the two variable names that are selected when the user clicks on one of the three plotting buttons. The function

- Gets the list of all items in the list box from the `String` property.
- Gets the indices of the selected items from the `Value` property.
- Returns two string variables, if there are two items selected. Otherwise `get_var_names` displays an error dialog explaining that the user must select two variables.

Here is the code for `get_var_names`:

```
function [var1,var2] = get_var_names(handles)
list_entries = get(handles.listbox1,'String');
index_selected = get(handles.listbox1,'Value');
if length(index_selected) ~= 2
   errordlg('You must select two variables',...
            'Incorrect Selection','modal')
else
   var1 = list_entries{index_selected(1)};
   var2 = list_entries{index_selected(2)};
end
```

Callbacks for the Plotting Buttons

The callbacks for the plotting buttons call `get_var_names` to get the names of the variables to plot and then call `evalin` to execute the plot commands in the base workspace.

For example, here is the callback for the `plot` function:

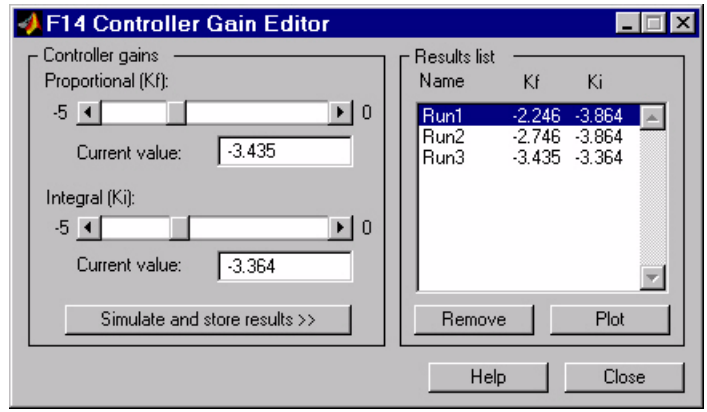
```
function plot_button_Callback(hObject, eventdata, handles)
[x,y] = get_var_names(handles);
evalin('base',['plot(' x ',' y ')'])
```

The command to evaluate is created by concatenating the strings and variables that result in the command:

```
plot(x,y)
```

A GUI to Set Simulink Model Parameters

This example illustrates how to create a GUI that sets the parameters of a Simulink® model. In addition, the GUI can run the simulation and plot the results. The following picture shows the GUI after running three simulations with different values for controller gains.



Techniques Used in This Example

This example illustrates a number of GUI building techniques:

- Opening and setting parameters on a Simulink model from a GUI.
- Implementing sliders that operate in conjunction with text boxes, which display the current value as well as accepting user input.
- Enabling and disabling controls, depending on the state of the GUI.
- Managing a variety of shared data using the `handles` structure.
- Directing graphics output to figures with hidden handles.
- Adding a help button that displays `.html` files in the MATLAB Help browser.

View Completed Layout and Its GUI M-File

If you are reading this in the MATLAB Help browser, you can click the following links to display the GUIDE Layout Editor and the MATLAB Editor with a completed version of this example. This enables you to see the values of

all component properties and to understand how the components are assembled to create the GUI. You can also see a complete listing of the code that is discussed in the following sections.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

- Click here to display this GUI in the Layout Editor.
- Click here to display the GUI M-file in the editor.

How to Use the GUI (Text of GUI Help)

You can use the F14 Controller Gain Editor to analyze how changing the gains used in the Proportional-Integral Controller affect the aircraft's angle of attack and the amount of G force the pilot feels.

Note that the Simulink diagram `f14.mdl` must be open to run this GUI. If you close the F14 Simulink model, the GUI reopens it whenever it requires the model to execute.

Changing the Controller Gains

You can change gains in two blocks:

- The Proportional gain (Kf) in the Gain block
- The Integral gain (Ki) in the Transfer Function block

You can change either of the gains in one of the two ways:

- Move the slider associated with that gain.
- Type a new value into the **Current value** edit field associated with that gain.

The block's values are updated as soon as you enter the new value in the GUI.

Running the Simulation

Once you have set the gain values, you can run the simulation by clicking the **Simulate and store results** button. The simulation time and output vectors are stored in the **Results list**.

Plotting the Results

You can generate a plot of one or more simulation results by selecting the row of results (`Run1`, `Run2`, etc.) in the **Results list** that you want to plot and clicking the **Plot** button. If you select multiple rows, the graph contains a plot of each result.

The graph is displayed in a figure, which is cleared each time you click the **Plot** button. The figure's handle is hidden so that only the GUI can display graphs in this window.

Removing Results

To remove a result from the **Results list**, select the row or rows you want to remove and click the **Remove** button.

Running the GUI

The GUI is nonblocking and nonmodal since it is designed to be used as an analysis tool.

GUI Options Settings

This GUI uses the following GUI option settings:

- Resize behavior: **Non-resizable**
- Command-line accessibility: **Off**
- M-file options selected:
  - Generate callback function prototypes
  - GUI allows only one instance to run

Opening the Simulink Block Diagrams

This example is designed to work with the F14 Simulink model. Since the GUI sets parameters and runs the simulation, the F14 model must be open when the GUI is displayed. When the GUI M-file runs the GUI, it executes the `model_open` subfunction. The purpose of the subfunction is to

- Determine if the model is open (`find_system`).
- Open the block diagram for the model and the subsystem where the parameters are being set, if not open already (`open_system`).
- Change the size of the controller Gain block so it can display the gain value (`set_param`).
- Bring the GUI forward so it is displayed on top of the Simulink diagrams (`figure`).
- Set the block parameters to match the current settings in the GUI.

Here is the code for the `model_open` subfunction.

```
function model_open(handles)
if  isempty(find_system('Name','f14')),
   open_system('f14'); open_system('f14/Controller')
   set_param('f14/Controller/Gain','Position',[275 14 340 56])
   figure(handles.F14ControllerEditor)
   set_param('f14/Controller Gain','Gain',...
       get(handles.KfCurrentValue,'String'))
   set_param(...
      'f14/Controller/Proportional plus integral compensator',...
       'Numerator',...
       get(handles.KiCurrentValue,'String'))
end
```

Programming the Slider and Edit Text Components

This GUI employs a useful combination of components in its design. Each slider is coupled to an edit text component so that:

- The edit text displays the current value of the slider.
- The user can enter a value into the edit text box and cause the slider to update to that value.
- Both components update the appropriate model parameters when activated by the user.

Slider Callback

The GUI uses two sliders to specify block gains since these components enable the selection of continuous values within a specified range. When a user changes the slider value, the callback executes the following steps:

- Calls `model_open` to ensure that the Simulink model is open so that simulation parameters can be set.
- Gets the new slider value.
- Sets the value of the **Current value** edit text component to match the slider.
- Sets the appropriate block parameter to the new value (`set_param`).

Here is the callback for the **Proportional (Kf)** slider.

```
function KfValueSlider_Callback(hObject, eventdata, handles)
% Ensure model is open.
model_open(handles)
% Get the new value for the Kf Gain from the slider.
NewVal = get(hObject, 'Value');
% Set the value of the KfCurrentValue to the new value
% set by slider.
set(handles.KfCurrentValue,'String',NewVal)
% Set the Gain parameter of the Kf Gain Block to the new value.
set_param('f14/Controller/Gain','Gain',num2str(NewVal))
```

Note that, while a slider returns a number and the edit text requires a string, uicontrols automatically convert the values to the correct type.

The callback for the **Integral (Ki)** slider follows a similar approach.

Current Value Edit Text Callback

The edit text box enables users to type in a value for the respective parameter. When the user clicks on another component in the GUI after typing into the text box, the edit text callback executes the following steps:

- Calls `model_open` to ensure that the Simulink model is open so that it can set simulation parameters.
- Converts the string returned by the edit box `String` property to a double (`str2double`).
- Checks whether the value entered by the user is within the range of the slider:

  If the value is out of range, the edit text `String` property is set to the value of the slider (rejecting the number typed in by the user).

  If the value is in range, the slider `Value` property is updated to the new value.
- Sets the appropriate block parameter to the new value (`set_param`).

Here is the callback for the Kf **Current value** text box.

```
function KfCurrentValue_Callback(hObject, eventdata, handles)
% Ensure model is open.
model_open(handles)
% Get the new value for the Kf Gain.
NewStrVal = get(hObject, 'String');
NewVal = str2double(NewStrVal);
% Check that the entered value falls within the allowable range.
if  isempty(NewVal) || (NewVal< -5) || (NewVal>0),
   % Revert to last value, as indicated by KfValueSlider.
   OldVal = get(handles.KfValueSlider,'Value');
   set(hObject, 'String',OldVal)
else % Use new Kf value
   % Set the value of the KfValueSlider to the new value.
   set(handles.KfValueSlider,'Value',NewVal)
   % Set the Gain parameter of the Kf Gain Block
   % to the new value.
   set_param('f14/Controller/Gain','Gain',NewStrVal)
end
```

The callback for the Ki **Current value** follows a similar approach.

Running the Simulation from the GUI

The GUI **Simulate and store results** button callback runs the model simulation and stores the results in the `handles` structure. Storing data in the `handles` structure simplifies the process of passing data to other subfunction since this structure can be passed as an argument.

When a user clicks on the **Simulate and store results** button, the callback executes the following steps:

- Calls `sim`, which runs the simulation and returns the data that is used for plotting.
- Creates a structure to save the results of the simulation, the current values of the simulation parameters set by the GUI, and the run name and number.
- Stores the structure in the `handles` structure.
- Updates the list box `String` to list the most recent run.

Here is the **Simulate and store results** button callback.

```
function SimulateButton_Callback(hObject, eventdata, handles)
[timeVector,stateVector,outputVector] = sim('f14');
% Retrieve old results data structure
if isfield(handles,'ResultsData') &
~isempty(handles.ResultsData)
   ResultsData = handles.ResultsData;
   % Determine the maximum run number currently used.
   maxNum = ResultsData(length(ResultsData)).RunNumber;
   ResultNum = maxNum+1;
else % Set up the results data structure
   ResultsData = struct('RunName',[],'RunNumber',[],...
                 'KiValue',[],'KfValue',[],'timeVector',[],...
                 'outputVector',[]);
   ResultNum = 1;
end
if isequal(ResultNum,1),
   % Enable the Plot and Remove buttons
   set([handles.RemoveButton,handles.PlotButton],'Enable','on')
end
% Get Ki and Kf values to store with the data and put in the
results list.
Ki = get(handles.KiValueSlider,'Value');
Kf = get(handles.KfValueSlider,'Value');
ResultsData(ResultNum).RunName = ['Run',num2str(ResultNum)];
ResultsData(ResultNum).RunNumber = ResultNum;
ResultsData(ResultNum).KiValue = Ki;
ResultsData(ResultNum).KfValue = Kf;
ResultsData(ResultNum).timeVector = timeVector;
ResultsData(ResultNum).outputVector = outputVector;
% Build the new results list string for the listbox
ResultsStr = get(handles.ResultsList,'String');
if isequal(ResultNum,1)
   ResultsStr = {['Run1',num2str(Kf),' ',num2str(Ki)]};
else
   ResultsStr = [ResultsStr;...
   {['Run',num2str(ResultNum),' ',num2str(Kf),' ', ...
   num2str(Ki)]}];
end
```

```
set(handles.ResultsList,'String',ResultsStr);
% Store the new ResultsData
handles.ResultsData = ResultsData;
guidata(hObject, handles)
```

Removing Results from the List Box

The GUI **Remove** button callback deletes any selected item from the **Results list** list box. It also deletes the corresponding run data from the `handles` structure. When a user clicks on the **Remove** button, the callback executes the following steps:

- Determines which list box items are selected when a user clicks on the **Remove** button and removes these items from the list box `String` property by setting each item to the empty matrix `[]`.
- Removes the deleted data from the `handles` structure.
- Displays the string `<empty>` and disables the **Remove** and **Plot** buttons (using the `Enable` property), if all the items in the list box are removed.
- Save the changes to the `handles` structure (`guidata`).

Here is the **Remove** button callback.

```
function RemoveButton_Callback(hObject, eventdata, handles)
currentVal = get(handles.ResultsList,'Value');
resultsStr = get(handles.ResultsList,'String');
numResults = size(resultsStr,1);
% Remove the data and list entry for the selected value
resultsStr(currentVal) =[];
handles.ResultsData(currentVal)=[];
% If there are no other entries, disable the Remove and Plot button
% and change the list string to <empty>
if isequal(numResults,length(currentVal)),
   resultsStr = {'<empty>'};
   currentVal = 1;

set([handles.RemoveButton,handles.PlotButton],'Enable','off')
end
% Ensure that list box Value is valid, then reset Value and String
currentVal = min(currentVal,size(resultsStr,1));
set(handles.ResultsList,'Value',currentVal,'String',resultsStr)
```

```
% Store the new ResultsData
guidata(hObject, handles)
```

Plotting the Results Data

The GUI **Plot** button callback creates a plot of the run data and adds a legend. The data to plot is passed to the callback in the `handles` structure, which also contains the gain settings used when the simulation ran. When a user clicks on the **Plot** button, the callback executes the following steps:

- Collects the data for each run selected in the **Results list**, including two variables (time vector and output vector) and a color for each result run to plot.
- Generates a string for the legend from the stored data.
- Creates the figure and axes for plotting and saves the handles for use by the **Close** button callback.
- Plots the data, adds a legend, and makes the figure visible.

Plotting Into the Hidden Figure

The figure that contains the plot is created invisible and then made visible after adding the plot and legend. To prevent this figure from becoming the target for plotting commands issued at the command line or by other GUIs, its `HandleVisibility` and `IntegerHandle` properties are set to `off`. However, this means the figure is also hidden from the `plot` and `legend` commands.

Use the following steps to plot into a hidden figure:

- Save the handle of the figure when you create it.
- Create an axes, set its `Parent` property to the figure handle, and save the axes handle.
- Create the plot (which is one or more line objects), save these line handles, and set their `Parent` properties to the handle of the axes.
- Make the figure visible.

Plot Button Callback Listing

Here is the **Plot** button callback.

```
function PlotButton_Callback(hObject, eventdata, handles)
currentVal = get(handles.ResultsList,'Value');
```

```
% Get data to plot and generate command string with color
% specified
legendStr = cell(length(currentVal),1);
plotColor = {'b','g','r','c','m','y','k'};
for ctVal = 1:length(currentVal);
   PlotData{(ctVal*3)-2} =
handles.ResultsData(currentVal(ctVal)).timeVector;
   PlotData{(ctVal*3)-1} =
handles.ResultsData(currentVal(ctVal)).outputVector;
   numColor = ctVal - 7*( floor((ctVal-1)/7) );
   PlotData{ctVal*3} = plotColor{numColor};
   legendStr{ctVal} = ...
     [handles.ResultsData(currentVal(ctVal)).RunName,'; Kf=',...
     num2str(handles.ResultsData(currentVal(ctVal)).KfValue),...
      ';  Ki=', ...
      num2str(handles.ResultsData(currentVal(ctVal)).KiValue)];
end
% If necessary, create the plot figure and store in handles
% structure
if ~isfield(handles,'PlotFigure') ||...
   ~ishandle(handles.PlotFigure),
   handles.PlotFigure = ...
        figure('Name','F14 Simulation Output',...
        'Visible','off','NumberTitle','off',...
        'HandleVisibility','off','IntegerHandle','off');
   handles.PlotAxes = axes('Parent',handles.PlotFigure);
   guidata(hObject, handles)
end
% Plot data
pHandles = plot(PlotData{:},'Parent',handles.PlotAxes);
% Add a legend, and bring figure to the front
legend(pHandles(1:2:end),legendStr{:})
% Make the figure visible and bring it forward
figure(handles.PlotFigure)
```

The GUI Help Button

The GUI **Help** button callback displays an HTML file in the MATLAB Help browser. It uses two commands:

- The `which` command returns the full path to the file when it is on the MATLAB path
- The `web` command displays the file in the Help browser.

This is the **Help** button callback.

```
function HelpButton_Callback(hObject, eventdata, handles)
HelpPath = which('f14ex_help.html');
web(HelpPath);
```

You can also display the help document in a Web browser or load an external URL. See the Web documentation for a description of these options.

Closing the GUI

The GUI **Close** button callback closes the plot figure, if one exists and then closes the GUI. The handle of the plot figure and the GUI figure are available from the `handles` structure. The callback executes two steps:

- Checks to see if there is a `PlotFigure` field in the `handles` structure and if it contains a valid figure handle (the user could have closed the figure manually).
- Closes the GUI figure

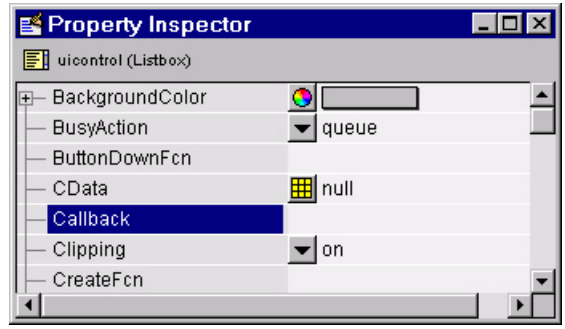
This is the **Close** button callback.

```
function CloseButton_Callback(hObject, eventdata, handles)
% Close the GUI and any plot window that is open
if isfield(handles,'PlotFigure') && ...
      ishandle(handles.PlotFigure),
   close(handles.PlotFigure);
end
close(handles.F14ControllerEditor);
```

The List Box Callback and Create Function

This GUI does not use the list box callback since the actions performed on list box items are carried out by push buttons (**Simulate and store results**, **Remove**, and **Plot**). However, GUIDE automatically inserts a callback stub when you add the list box and automatically sets the `Callback` property to execute this subfunction whenever the callback is triggered (which happens when users select an item in the list box).

In this case, there is no need for the list box callback to execute, so you should delete it from the GUI M-file. It is important to remember to also delete the `Callback` property string so MATLAB does not attempt to execute the callback. You can do this using the property inspector:



See the description of list boxes for more information on how to trigger the list box callback.

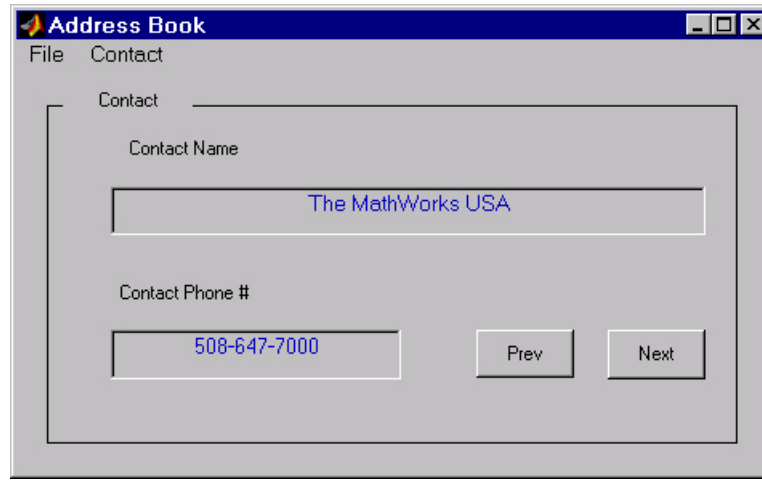
Setting the Background to White

The list box create function enables you to determine the background color of the list box. The following code shows the create function for the list box that is tagged `ResultsList`.

```
function ResultsList_CreateFcn(hObject, eventdata, handles)
% Hint: listbox controls usually have a white background, change
%       'usewhitebg' to 0 to use default. See ISPC and COMPUTER.
usewhitebg = 1;
if usewhitebg
    set(hObject,'BackgroundColor','white');
else
set(hObject,'BackgroundColor',...
    get(0,'defaultUicontrolBackgroundColor'));
end
```

An Address Book Reader

This example shows how to implement a GUI that displays names and phone numbers, which it reads from a MAT-file.



Techniques Used in This Example

This example demonstrates the following GUI programming techniques:

- Uses open and save dialogs to provide a means for users to locate and open the address book MAT-files and to save revised or new address book MAT-files.
- Defines callbacks written for GUI menus.
- Uses the GUI's `handles` structure to save and recall shared data.
- Uses a GUI figure resize function.

Managing Shared Data

One of the key techniques illustrated in this example is how to keep track of information and make it available to the various subfunctions. This information includes

- The name of the current MAT-file
- The names and phone numbers stored in the MAT-file

- An index pointer that indicates the current name and phone number, which must be updated as the user pages through the address book
- The figure position and size
- The handles of all GUI components

The descriptions of the subfunctions that follow illustrate how to save and retrieve information from the `handles` structure. See "The Handles Structure" on page 8-13 for background information on this structure.

View Completed Layout and Its GUI M-File

If you are reading this in the MATLAB Help browser, you can click the following links to display the GUIDE Layout Editor and the MATLAB Editor with a completed version of this example. This enables you to see the values of all component properties and to understand how the components are assembled to create the GUI. You can also see a complete listing of the code that is discussed in the following sections.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

- Click here to display this GUI in the Layout Editor.
- Click here to display the GUI M-file in the MATLAB Editor.

Running the GUI

The GUI is nonblocking and nonmodal since it is designed to be displayed while you perform other MATLAB tasks.

GUI Option Settings

This GUI uses the following GUI option settings:

- Resize behavior: **User-specified**
- Command-line accessibility: **Off**

- GUI M-file options selected:
  - Generate callback function prototypes
  - Application allows only one instance to run

Calling the GUI

You can call the GUI M-file with no arguments, in which case the GUI uses the default address book MAT-file, or you can specify an alternate MAT-file from which the GUI reads information. In this example, the user calls the GUI with a pair of arguments, `address_book('book', 'my_list.mat')`. The first argument, `'book'`, is a key word that the M-file looks for in the opening function. If the M-file finds the key word, it knows to use the second argument as the MAT-file for the address book. Calling the GUI with this syntax is analogous to calling it with a valid property-value pair, such as `('color', 'red')`. However, since `'book'` is not a valid figure property, in this example the opening function in the M-file includes code to recognize the pair `('book', 'my_list.mat')`.

Note that it is not necessary to use the key word `'book'`. You could program the M-file to accept just the MAT-file as an argument, using the syntax `address_book('my_list.mat')`. The advantage of calling the GUI with the pair `('book', 'my_list.mat')` is that you can program the GUI to accept other user arguments, as well as valid figure properties, using the property-value pair syntax. The GUI can then identify which property the user wants to specify from the property name.

The following code shows how to program the opening function to look for the key word `'book'`, and if it finds the key word, to use the MAT-file specified by the second argument as the list of contacts.

```
function address_book_OpeningFcn(hObject, eventdata, ...
    handles, varargin)
% Choose default command line output for address_book
handles.output = hObject;
% Update handles structure
guidata(hObject, handles);
% User added code follows
if nargin < 4
    % Load the default address book
    Check_And_Load([],handles);
    % If the first element in varargin is 'book' and
```

```
    & the second element is a MATLAB file, then load that file
elseif (length(varargin) == 2 && ...
       strcmpi(varargin{1},'book') && ...
       (2 == exist(varargin{2},'file')))
    Check_And_Load(varargin{2},handles);
else
    errordlg('File Not Found','File Load Error')
    set(handles.Contact_Name,'String','')
    set(handles.Contact_Phone,'String','')
end
```

Loading an Address Book Into the Reader

There are two ways in which an address book (i.e., a MAT-file) is loaded into the GUI:

- When running the GUI, you can specify a MAT-file as an argument. If you do not specify an argument, the GUI loads the default address book (`addrbook.mat`).
- The user can select **Open** under the **File** menu to browse for other MAT-files.

Validating the MAT-file

To be a valid address book, the MAT-file must contain a structure called `Addresses` that has two fields called `Name` and `Phone`. The `Check_And_Load` subfunction validates and loads the data with the following steps:

- Loads (`load`) the specified file or the default if none is specified.
- Determines if the MAT-file is a valid address book.
- Displays the data if it is valid. If it is not valid, displays an error dialog (`errordlg`).
- Returns `1` for valid MAT-files and `0` if invalid (used by the **Open** menu callback)
- Saves the following items in the `handles` structure:
  - The name of the MAT-file
  - The `Addresses` structure
  - An index pointer indicating which name and phone number are currently displayed

Check_And_Load Code Listing

This is the Check_And_Load function.

```
function pass = Check_And_Load(file,handles)
% Initialize the variable "pass" to determine if this is a valid
% file.
pass = 0;
% If called without any file then set file to the default
% filename.
% Otherwise, if the file exists then load it.
if isempty(file)
   file = 'addrbook.mat';
   handles.LastFile = file;
   guidata(handles.Address_Book,handles)
end
if exist(file) == 2
   data = load(file);
end
% Validate the MAT-file
% The file is valid if the variable is called "Addresses" and
% it has fields called "Name" and "Phone"
flds = fieldnames(data);
if (length(flds) == 1) && (strcmp(flds{1},'Addresses'))
   fields = fieldnames(data.Addresses);
   if (length(fields) == 2) && ...
         (strcmp(fields{1},'Name')) && ...
         (strcmp(fields{2},'Phone'))
       pass = 1;
   end
end
% If the file is valid, display it
if pass
   % Add Addresses to the handles structure
   handles.Addresses = data.Addresses;
   guidata(handles.Address_Book,handles)
   % Display the first entry
   set(handles.Contact_Name,'String',data.Addresses(1).Name)
   set(handles.Contact_Phone,'String',data.Addresses(1).Phone)
   % Set the index pointer to 1 and save handles
   handles.Index = 1;
```

```
    guidata(handles.Address_Book,handles)
else
    errordlg('Not a valid Address Book','Address Book Error')
end
```

The Open Menu Callback

The address book GUI contains a **File** menu that has an **Open** submenu for loading address book MAT-files. When selected, **Open** displays a dialog (`uigetfile`) that enables the user to browse for files. The dialog displays only MAT-files, but users can change the filter to display all files.

The dialog returns both the filename and the path to the file, which is then passed to `fullfile` to ensure the path is properly constructed for any platform. `Check_And_Load` validates and load the new address book.

Open_Callback Code Listing

```
function Open_Callback(hObject, eventdata, handles)
[filename, pathname] = uigetfile( ...
    {'*.mat', 'All MAT-Files (*.mat)'; ...
        '*.*','All Files (*.*)'}, ...
    'Select Address Book');
% If "Cancel" is selected then return
if isequal([filename,pathname],[0,0])
    return
% Otherwise construct the fullfilename and Check and load the file
else
    File = fullfile(pathname,filename);
    % if the MAT-file is not valid, do not save the name
    if Check_And_Load(File,handles)
        handles.LastFIle = File;
        guidata(hObject, handles)
    end
end
```

See the “Creating Menus” on page 6-48 section for information on creating the menu.

The Contact Name Callback

The **Contact Name** text box displays the name of the address book entry. If you type in a new name and press enter, the callback performs these steps:

- If the name exists in the current address book, the corresponding phone number is displayed.
- If the name does not exist, a question dialog (`questdlg`) asks you if you want to create a new entry or cancel and return to the name previously displayed.
- If you create a new entry, you must save the MAT-file with the **File** -> **Save** menu.

Storing and Retrieving Data

This callback makes use of the `handles` structure to access the contents of the address book and to maintain an index pointer (`handles.Index`) that enables the callback to determine what name was displayed before it was changed by the user. The index pointer indicates what name is currently displayed. The address book and index pointer fields are added by the `Check_And_Load` function when the GUI is run.

If the user adds a new entry, the callback adds the new name to the address book and updates the index pointer to reflect the new value displayed. The updated address book and index pointer are again saved (`guidata`) in the `handles` structure.

Contact Name Callback

```
function Contact_Name_Callback(hObject, eventdata, handles)
% Get the strings in the Contact Name and Phone text box
Current_Name = get(handles.Contact_Name,'string');
Current_Phone = get(handles.Contact_Phone,'string');
% If empty then return
if isempty(Current_Name)
   return
end
% Get the current list of addresses from the handles structure
Addresses = handles.Addresses;
% Go through the list of contacts
% Determine if the current name matches an existing name
for i = 1:length(Addresses)
   if strcmp(Addresses(i).Name,Current_Name)
```

```
        set(handles.Contact_Name,'string',Addresses(i).Name)
        set(handles.Contact_Phone,'string',Addresses(i).Phone)
        handles.Index = i;
        guidata(hObject, handles)
        return
    end
end
% If it's a new name, ask to create a new entry
Answer=questdlg('Do you want to create a new entry?', ...
    'Create New Entry', ...
    'Yes','Cancel','Yes');
switch Answer
case 'Yes'
    Addresses(end+1).Name = Current_Name; % Grow array by 1
    Addresses(end).Phone = Current_Phone;
    index = length(Addresses);
    handles.Addresses = Addresses;
    handles.Index = index;
    guidata(hObject, handles)
    return
case 'Cancel'
    % Revert back to the original number

set(handles.Contact_Name,'String',Addresses(handles.Index).Name)

set(handles.Contact_Phone,'String',Addresses(handles.Index).Phon
e)
    return
end
```

The Contact Phone Number Callback

The **Contact Phone #** text box displays the phone number of the entry listed in the **Contact Name** text box. If you type in a new number click one of the push buttons, the callback opens a question dialog that asks you if you want to change the existing number or cancel your change.

Like the **Contact Name** text box, this callback uses the index pointer (`handles.Index`) to update the new number in the address book and to revert to the previously displayed number if the user selects **Cancel** from the question

dialog. Both the current address book and the index pointer are saved in the `handles` structure so that this data is available to other callbacks.

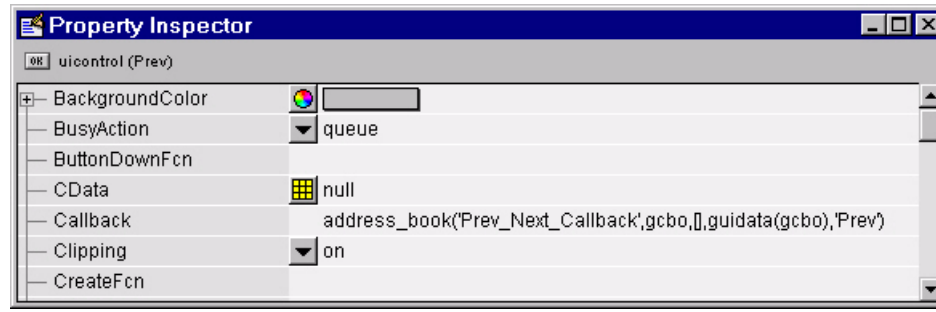
If you create a new entry, you must save the MAT-file with the **File –> Save** menu.

Code Listing

```
function Contact_Phone_Callback(hObject, eventdata, handles)
Current_Phone = get(handles.Contact_Phone,'string');
% If either one is empty then return
if isempty(Current_Phone)
   return
end
% Get the current list of addresses from the handles structure
Addresses = handles.Addresses;
Answer=questdlg('Do you want to change the phone number?', ...
   'Change Phone Number', ...
   'Yes','Cancel','Yes');
switch Answer
case 'Yes'
   % If no name match was found create a new contact
   Addresses(handles.Index).Phone = Current_Phone;
   handles.Addresses = Addresses;
   guidata(hObject, handles)
   return
case 'Cancel'
   % Revert back to the original number
   set(handles.Contact_Phone,...
       'String',Addresses(handles.Index).Phone)
   return
end
```

Paging Through the Address Book — Prev/Next

The **Prev** and **Next** buttons page back and forth through the entries in the address book. Both push buttons use the same callback, `Prev_Next_Callback`. You must set the `Callback` property of both push buttons to call this subfunction, as the following illustration of the **Prev** push button `Callback` property setting shows.



Determining Which Button Is Clicked

The callback defines an additional argument, `str`, that indicates which button, **Prev** or **Next**, was clicked. For the **Prev** button `Callback` property (illustrated above), the `Callback` string includes `'Prev'` as the last argument. The **Next** button `Callback` string includes `'Next'` as the last argument. The value of `str` is used in `case` statements to implement each button's functionality (see the code listing below).

Paging Forward or Backward

`Prev_Next_Callback` gets the current index pointer and the addresses from the `handles` structure and, depending on which button the user presses, the index pointer is decremented or incremented and the corresponding address and phone number are displayed. The final step stores the new value for the index pointer in the `handles` structure and saves the updated structure using `guidata`.

Code Listing

```
function Prev_Next_Callback(hObject, eventdata,handles,str)
% Get the index pointer and the addresses
index = handles.Index;
Addresses = handles.Addresses;
% Depending on whether Prev or Next was clicked,
% change the display
switch str
case 'Prev'
   % Decrease the index by one
   i = index - 1;
```

```
    % If the index is less than one then set it equal to the index
    %  of the last element in the Addresses array
    if i < 1
        i = length(Addresses);
    end
case 'Next'
    % Increase the index by one
    i = index + 1;
    % If the index is greater than the size of the array then
    % point to the first item in the Addresses array
    if i > length(Addresses)
        i = 1;
    end
end
% Get the appropriate data for the index in selected
Current_Name = Addresses(i).Name;
Current_Phone = Addresses(i).Phone;
set(handles.Contact_Name,'string',Current_Name)
set(handles.Contact_Phone,'string',Current_Phone)
% Update the index pointer to reflect the new index
handles.Index = i;
guidata(hObject, handles)
```

Saving Changes to the Address Book from the Menu

When you make changes to an address book, you need to save the current MAT-file, or save it as a new MAT-file. The **File** submenus **Save** and **Save As** enable you to do this. These menus, created with the Menu Editor, use the same callback, `Save_Callback`.

The callback uses the menu `Tag` property to identify whether **Save** or **Save As** is the callback object (i.e., the object whose handle is passed in as the first argument to the callback function). You specify the menu's `Tag` property with the Menu Editor.

Saving the Addresses Structure

The `handles` structure contains the `Addresses` structure, which you must save (`handles.Addresses`) as well as the name of the currently loaded MAT-file (`handles.LastFile`). When the user makes changes to the name or number,

the `Contact_Name_Callback` or the `Contact_Phone_Callback` updates `handles.Addresses`.

Saving the MAT-File

If the user selects **Save**, the `save` command is called to save the current MAT-file with the new names and phone numbers.

If the user selects **Save As**, a dialog is displayed (`uiputfile`) that enables the user to select the name of an existing MAT-file or specify a new file. The dialog returns the selected filename and path. The final steps include

- Using `fullfile` to create a platform-independent pathname.
- Calling `save` to save the new data in the MAT-file.
- Updating the `handles` structure to contain the new MAT-file name.
- Calling `guidata` to save the `handles` structure.

Save_Callback Code Listing

```
function Save_Callback(hObject, eventdata, handles)
% Get the Tag of the menu selected
Tag = get(hObject, 'Tag');
% Get the address array
Addresses = handles.Addresses;
% Based on the item selected, take the appropriate action
switch Tag
case 'Save'
   % Save to the default addrbook file
   File = handles.LastFile;
   save(File,'Addresses')
case 'Save_As'
   % Allow the user to select the file name to save to
   [filename, pathname] = uiputfile( ...
       {'*.mat';'*.*'}, ...
       'Save as');
   % If 'Cancel' was selected then return
   if isequal([filename,pathname],[0,0])
       return
   else
       % Construct the full path and save
       File = fullfile(pathname,filename);
```

```
        save(File,'Addresses')
        handles.LastFile = File;
        guidata(hObject, handles)
    end
end
```

The Create New Menu

The **Create New** menu simply clears the **Contact Name** and **Contact Phone #** text fields to facilitate adding a new name and number. After making the new entries, the user must then save the address book with the **Save** or **Save As** menus. This callback sets the text `String` properties to empty strings:

```
function New_Callback(hObject, eventdata, handles)
set(handles.Contact_Name,'String','')
set(handles.Contact_Phone,'String','')
```

The Address Book Resize Function

The address book defines its own resize function. To use this resize function, you must set the **Application Options** dialog **Resize behavior** to `User-specified`, which in turn sets the figure's `ResizeFcn` property to:

```
address_book('ResizeFcn',gcbo,[],guidata(gcbo))
```

Whenever the user resizes the figure, MATLAB calls the `ResizeFcn` subfunction in the address book M-file (`address_book.m`)

Behavior of the Resize Function

The `resize` function allows users to make the figure wider, to accommodate long names and numbers, but does not allow the figure to be made narrower than its original width. Also, users cannot change the height. These restrictions do not limit the usefulness of the GUI and simplify the `resize` function, which must maintain the proper proportions between the figure size and the components in the GUI.

When the user resizes the figure and releases the mouse, the resize function executes. At that point, the resized figure's dimensions are saved. The following sections describe how the `resize` function handles the various possibilities.

Changing the Width

If the new width is greater than the original width, set the figure to the new width.

The size of the **Contact Name** text box changes in proportion to the new figure width. This is accomplished by:

- Changing the `Units` of the text box to `normalized`.
- Resetting the width of the text box to be 78.9% of the figure's width.
- Returning the `Units` to `characters`.

If the new width is less than the original width, use the original width.

Changing the Height

If the user attempts to change the height, use the original height. However, because the resize function is triggered when the user releases the mouse button after changing the size, the resize function cannot always determine the original position of the GUI on screen. Therefore, the resize function applies a compensation to the vertical position (second element in the figure `Position` vector) as follows:

> vertical position when mouse released + height when mouse released minus the original height

When the figure is resized from the bottom, it stays in the same position. When resized from the top, the figure moves to the location where the mouse button is released.

Ensuring the Resized Figure Is On Screen

The `resize` function calls `movegui` to ensure that the resized figure is on screen regardless of where the user releases the mouse.

When the GUI is first run, it is displayed at the size and location specified by the figure `Position` property. You can set this property with the Property Inspector when you create the GUI.

Code Listing

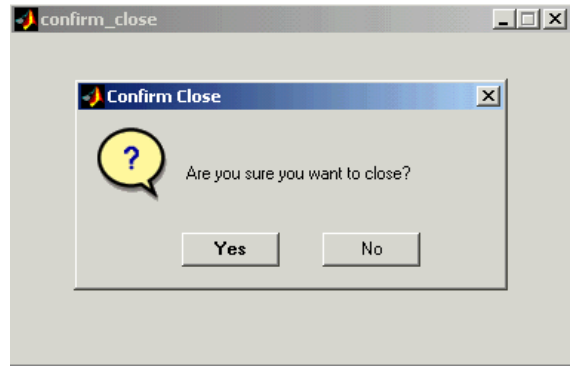
```
function ResizeFcn(hObject, eventdata, handles)
% Get the figure size and position
Figure_Size = get(hObject, 'Position');
% Set the figure's original size in character units
```

```
Original_Size = [ 0 0 94 19.230769230769234];
% If the resized figure is smaller than the
% original figure size then compensate.
if (Figure_Size(3)<Original_Size(3)) | ...
   (Figure_Size(4) ~= Original_Size(4))
   if Figure_Size(3) < Original_Size(3)
       % If the width is too small then reset to origianl width.
       set(hObject, 'Position',...
           [Figure_Size(1), Figure_Size(2), ...
            Original_Size(3), Original_Size(4)])
       Figure_Size = get(hObject, 'Position');
   end
   if Figure_Size(4) ~= Original_Size(4)
       % Do not allow the height to change.
       set(hObject, 'Position',...
           [Figure_Size(1),...
            Figure_Size(2)+Figure_Size(4)-Original_Size(4),...
            Figure_Size(3), Original_Size(4)])
   end
end
% Adjust the size of the Contact Name text box.
% Set the units of the Contact Name field to 'Normalized'.
set(handles.Contact_Name,'units','normalized')
% Get its Position.
C_N_pos = get(handles.Contact_Name,'Position');
% Reset it so that it's width remains normalized.
% relative to figure.
set(handles.Contact_Name,'Position',...
   [C_N_pos(1) C_N_pos(2)  0.789 C_N_pos(4)])
% Return the units to 'Characters'.
set(handles.Contact_Name,'units','characters')
% Reposition GUI on screen.
movegui(hObject, 'onscreen')
```

Using a Modal Dialog to Confirm an Operation

This example illustrates how to use the modal dialog GUI together with another GUI that has a **Close** button. Clicking the **Close** button displays the modal dialog, which asks users to confirm that they really want to proceed with the close operation.

The following figure illustrates the dialog positioned over the GUI application, awaiting the user's response.



The example is presented in the following sections:

- "View Completed Layouts and Their GUI M-Files" on page 10-47
- "Setting Up the Close Confirmation Dialog" on page 10-48
- "Setting Up the GUI with the Close Button" on page 10-49
- "Running the GUI with the Close Button" on page 10-50
- "How the GUI and Dialog Work" on page 10-51

View Completed Layouts and Their GUI M-Files

If you are reading this in the MATLAB Help Browser, you can click the following links to display the GUIDE Layout Editor and the MATLAB Editor with a completed version of this example. This enables you to see the values of all component properties and to understand how the components are assembled to create the GUI. You can also see a complete listing of the code that is discussed in the following sections.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

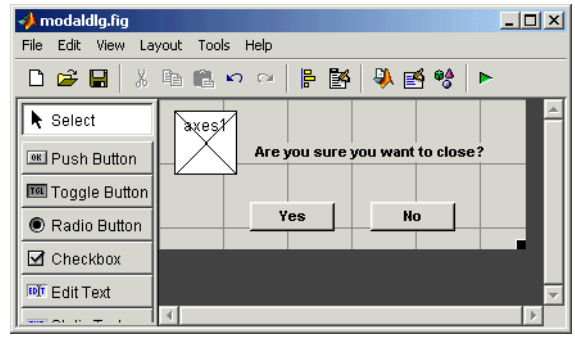
- Click here to display the GUIs in the Layout Editor.
- Click here to display the GUI M-files in the editor.

Setting Up the Close Confirmation Dialog

To set up the dialog, do the following:

1. Select **New** from the **File** menu in the GUIDE Layout Editor.
2. In the **GUIDE Quick Start** dialog, select the **Modal Question Dialog** template and click **OK**.
3. Right-click the static text, `Do you want to create a question dialog?`, in the Layout Editor and select **Property Inspector** from the pop-up menu.
4. Scroll down to `String` in the Property Inspector and change the `String` property to `Are you sure you want to close?`
5. Select **Save** from the **File** menu and type `modaldlg.fig` in the **File name** field.

The GUI should now appear as in the following figure.



Setting Up the GUI with the Close Button

To set up the second GUI with a **Close** button, do the following:

1. Select **New** from the **File** menu in the GUIDE Layout Editor.
2. In the **GUIDE Quick Start** dialog, select **Blank GUI (Default)** and click **OK**. This opens the blank GUI in a new Layout Editor window.
3. Drag a push button from the Component palette of the Layout Editor into the layout area.
4. Right-click the push button and select **Property Inspector** from the pop-up menu.
5. Change the `String` property to `Close`.
6. Change the `Tag` property to `close_pushbutton`.
7. Click the M-file editor icon on the toolbar of the Layout Editor.
8. Click the callback icon on the toolbar of the M-file editor and select `close_pushbutton_Callback` from the menu.

   The following generated code for the close button callback should appear in the M-file editor:

```
% --- Executes on button press in close_pushbutton.
function close_pushbutton_Callback(hObject, eventdata, handles)
% hObject    handle to close_pushbutton (see GCBO)
```

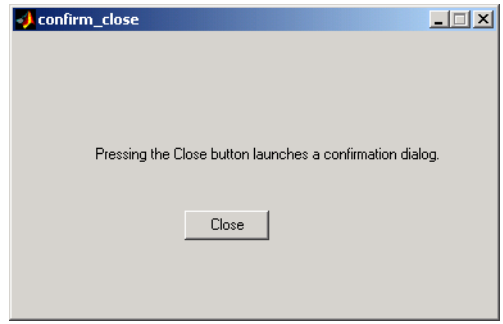
```
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

9 After these comments, add the following code:

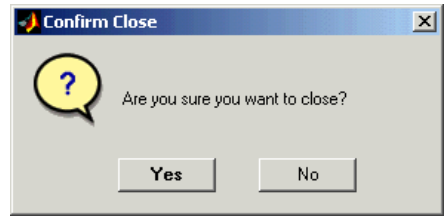
```
% Get the current position of the GUI from the handles structure
% to pass to the modal dialog.
pos_size = get(handles.figure1,'Position');
% Call modaldlg with the argument 'Position'.
user_response = modaldlg('Title','Confirm Close');
switch user_response
case {'No'}
   % take no action
case 'Yes'
   % Prepare to close GUI application window
   %                  .
   %                  .
   %                  .
   delete(handles.figure1)
end
```

Running the GUI with the Close Button

Run the GUI with the **Close** button by clicking the **Run** button on the Layout Editor toolbar. The GUI appears as in the following figure:



When you click the **Close** button on the GUI, the modal dialog appears as shown in the following figure:



Clicking the **Yes** button closes both the close dialog and the GUI that calls it. Clicking the **No** button closes just the dialog.

How the GUI and Dialog Work

This section describes what occurs when you click the **Close** button on the GUI:

1. User clicks the **Close** button. Its callback then
   - Gets the current position of the GUI from the `handles` structure with the command

     ```
     pos_size = get(handles.figure1,'Position')
     ```

   - Calls the modal dialog with the command

     ```
     user_response = modaldlg('Title','Confirm Close');
     ```

     This is an example of calling a GUI with a property value pair. In this case, the figure property is `'Title'`, and its value is the string `'Confirm Close'`. Opening `modaldlg` with this syntax displays the text “Confirm Close” at the top of the dialog.

2. The modal dialog opens with the `'Position'` obtained from the GUI that calls it.

3. The opening function in the modal dialog M-file:
   - Makes the dialog modal.
   - Executes the `uiwait` command, which causes the dialog to wait for the user to click the **Yes** button or the **No** button, or click the close box (`X`) on the window border.

4. When a user clicks one of the two push buttons, the callback for the push button

- Updates the output field in the `handles` structure
- Executes `uiresume` to return control to the opening function where `uiwait` is called.

5. The output function is called, which returns the string `Yes` or `No` as an output argument, and deletes the dialog with the command

   ```
   delete(handles.figure1)
   ```

6. When the GUI with the **Close** button regains control, it receives the string `Yes` or `No`. If the answer is `'No'`, it does nothing. If the answer is `'Yes'`, the close button callback closes the GUI with the command

   ```
   delete(handles.figure1)
   ```

Creating GUIs Programmatically

This section provides examples of graphical user interfaces (GUIs) created programmatically.

Functions for Creating GUIs (p. 11-1)	Summarizes MATLAB functions specifically intended for GUI creation. In creating your GUI, you can also use any other MATLAB or Toolbox functions appropriate to your GUI.
Examples of GUIs Created Programmatically (p. 12-7)	Provides three examples that illustrate the application of some programming techniques used to create GUIs.

Note Following the release of MATLAB version 7.2, Creating Graphical User Interfaces will be further updated and expanded. The HTML and PDF versions of this document will be updated on The MathWorks Web site some time after the release. Check this page and the title page of the PDF document to determine if they have been updated.

11

Functions for Creating GUIs

Introduction (p. 11-2)	Introduces the topic.
Function Summary (p. 11-3)	Summarizes MATLAB functions specifically intended for GUI creation.

Introduction

MATLAB provides a selection of functions for creating GUIs. Some of these create standard dialog boxes with just one function call. Others enable you to create customized GUIs that can be simple or complex.

This topic lists the functions that are specifically meant for GUI creation. In creating your GUI, you can also use any other MATLAB or Toolbox functions.appropriate to the domain of you GUI.

Function Summary

- "Predefined Dialog Boxes" on page 11-3
- "Creating Components" on page 11-3
- "ActiveX Controls" on page 11-4
- "Using Application-Defined Data" on page 11-4
- "Interactive User Input" on page 11-4
- "Working with Preferences" on page 11-4
- "Utilities" on page 11-5

Predefined Dialog Boxes

MATLAB provides a selection of standard dialog boxes that you can create with a single function call. For information about these dialog boxes and the functions used to create them, see Predefined Dialog Boxes in the MATLAB documentation.

Creating Components

`axes`	Axes object in which you can create plots, graphs, etc.	Properties
`uibuttongroup`	Container objects to exclusively manage radio buttons and toggle buttons	Properties
`uicontextmenu`	Context menus	Properties
`uicontrol`	User interface control objects, including check boxes, editable text fields, list boxes, pop-up menus, push buttons, radio buttons, sliders, static text labels, and toggle buttons	Properties
`uimenu`	Menus on figure windows	Properties
`uipanel`	Create panel container objects to group user interface controls, axes, and other container objects	Properties
`uipushtool`	Push buttons on a toolbar	Properties

`uitoggletool`	Toggle buttons on a toolbar	Properties
`uitoolbar`	Toolbar on figure	Properties

ActiveX Controls

`actxcontrol`	ActiveX control in a figure window.
`actxcontrollist`	List all currently installed ActiveX controls
`actxcontrolselect`	Display graphical interface for creating an ActiveX control
`actxserver`	Create COM Automation server

Using Application-Defined Data

`getappdata`	Value of application-defined data
`guidata`	Store or retrieve GUI data
`isappdata`	True if application-defined data exists
`rmappdata`	Remove application-defined data
`setappdata`	Specify application-defined data

Interactive User Input

`waitfor`	Wait for condition before resuming execution
`waitforbuttonpress`	Wait for key press or mouse button click
`ginput`	Graphical input from a mouse or cursor

Working with Preferences

`addpref`	Add preference
`getpref`	Preference
`ispref`	Test for existence of preference
`rmpref`	Remove preference

`setpref`	Set preference
`uigetpref`	Open dialog box for retrieving preferences
`uisetpref`	Manage preferences used in `uigetpref`

Utilities

`align`	Align uicontrol and axes
`findall`	Find all graphics objects
`findfigs`	Find visible off-screen figures
`findobj`	Locate graphics objects with specific properties
`gcbf`	Handle of figure containing object whose callback is executing
`gcbo`	Handle of object whose callback is executing
`guihandles`	Create structure of handles
`inspect`	Open Property Inspector
`movegui`	Move GUI figure to specified location on screen
`openfig`	Open new copy or raise existing copy of GUI figure
`selectmoveresize`	Select, move, resize, or copy axes and uicontrol graphics objects
`textwrap`	Return wrapped string matrix for given uicontrol
`uiresume`	Resume program execution halted with `uiwait`
`uistack`	Restack objects
`uiwait`	Halt program execution, restart with `uiresume`

12

Examples of GUIs Created Programmatically

Introduction (p. 12-8)	Introduces the examples and lists the programming techniques they illustrate.
GUI with Axes, Menu, and Toolbar (p. 12-9)	Creates a GUI that displays a user-selected plot in an axes.
Color Palette (p. 12-22)	Creates a color palette that can be embedded in a host GUI. The color palette enables a user to select colors.
Icon Editor (p. 12-33)	Creates an icon editor that enables a user to create and edit icons. It embeds the color palette from the previous example.

Introduction

This chapter provides three examples that illustrate the application of certain techniques in programmatically created GUIs.

- "GUI with Axes, Menu, and Toolbar" on page 12-9
- "Color Palette" on page 12-22
- "Icon Editor" on page 12-33

Each example lists the techniques it illustrates. These techniques include:

- Creation of a dialog that does not return until the user makes a choice
- Passing input arguments to the GUI when it is opened
- Obtaining output from the GUI when it returns
- Shielding the GUI from accidental changes
- Running the GUI across multiple platforms
- Making a GUI modal
- Sharing data among multiple GUIs
- Creating menus and context menus
- Creating toolbars
- Using an external utility function
- Achieving proper resize behavior

The examples all use nested functions. For information about using nested functions, see Nested Functions in the MATLAB documentation.

GUI with Axes, Menu, and Toolbar

The initial topics introduce the example and the techniques it illustrates:

- "The Example" on page 12-9
- "Techniques Used in the Example" on page 12-11
- "View the Completed GUI M-Files" on page 12-11

The remaining topics describe the example in the sequence in which the main M-file is organized:

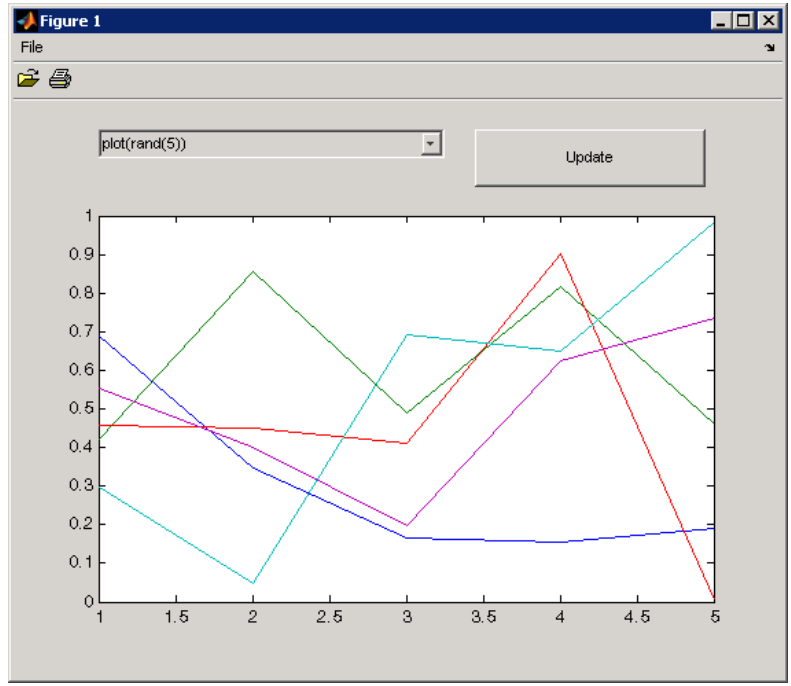
1. "Creating the Data" on page 12-11
2. "Creating the GUI and Its Components" on page 12-12
3. "Initializing the GUI" on page 12-17
4. "Defining the Callbacks" on page 12-18
5. "Helper Function: Plotting the Plot Types" on page 12-21

The Example

This example creates a GUI that displays a user-selected plot in an axes. The GUI contains the following components:

- Axes
- Pop-up menu with a list of five plots
- Push button for updating the contents of the axes
- Menu bar File menu with three items: Open, Print, and Close
- Toolbar with two buttons that enable a user to open files and print the plot.

When you run the GUI, it initially displays a plot of five random numbers generated by the MATLAB `rand(5)` command, as shown in the following figure.



You can select other plots in the pop-up menu. Clicking the **Update** button displays the currently selected plot on the axes.

The GUI **File** menu has three items:

- **Open** displays a dialog from which you can open files on your computer.
- **Print** opens the **Print** dialog. Clicking **Yes** in the **Print** dialog prints the plot.
- **Close** closes the GUI.

The GUI toolbar has two buttons:

- The Open button performs the same function as the **Open** menu item. It displays a dialog from which you can open files on your computer.
- The Print button performs the same function as the **Print** menu item. It opens the **Print** dialog. Clicking **Yes** in the **Print** dialog prints the plot.

Techniques Used in the Example

This example illustrates the following techniques:

- Passing input arguments to the GUI when it is opened
- Obtaining output from the GUI when it returns
- Shielding the GUI from accidental changes
- Running the GUI across multiple platforms
- Creating menus
- Creating toolbars
- Achieving proper resize behavior

Note This example uses nested functions. For information about using nested functions, see Nested Functions in the MATLAB documentation.

View the Completed GUI M-Files

If you are reading this in the MATLAB Help browser, you can click the following links to display the MATLAB Editor with complete listings of the code used in this example.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

- Click here to display the main GUI M-file in the MATLAB Editor.
- Click here to display the utility `iconRead` M-file in the MATLAB Editor.

Creating the Data

The example defines two variables `mOutputArgs` and `mPlotTypes`.

`mOutputArgs` is a cell array that holds output values should the user request them. It later assigns a default value to this argument.

```
mOutputArgs = {};  % Variable for storing output when GUI returns
```

`mPlotTypes` is a 5-by-2 cell array that holds the data to be plotted in the axes. The first column contains the strings that are used to populate the pop-up menu. The second column contains the functions, as anonymous function handles, that create the plots.

```
mPlotTypes = {...      % Example plot types shown by this GUI
              'plot(rand(5))',        @(a)plot(a,rand(5));
             'plot(sin(1:0.01:25))', @(a)plot(a,sin(1:0.01:25));
              'bar(1:.5:10)',         @(a)bar(a,1:.5:10);
              'plot(membrane)',       @(a)plot(a,membrane);
              'surf(peaks)',          @(a)surf(a,peaks)};
```

Because the data is created at the top level of the GUI function, it is available to all callbacks and other functions in the GUI.

See Anonymous Functions in the MATLAB documentation for information about using anonymous functions.

Creating the GUI and Its Components

Like the data, the components are created at the top level so that their handles are available to all callbacks and other functions in the GUI.

- "The Main Figure" on page 12-12
- "The Axes" on page 12-13
- "The Pop-Up Menu" on page 12-14
- "The Update Push Button" on page 12-14
- "The File Menu and Its Menu Items" on page 12-15
- "The Toolbar and Its Tools" on page 12-16

The Main Figure

The following statement creates the figure for GUI.

```
hMainFigure = figure(...       % The main GUI figure
                     'MenuBar','none', ...
                     'Toolbar','none', ...
                     'HandleVisibility','callback', ...
                     'Color', get(0,...
                             'defaultuicontrolbackgroundcolor'));
```

- The `figure` function creates the GUI figure.
- Setting the `MenuBar` and `Toolbar` properties to `none`, prevents the standard menu bar and toolbar from displaying.
- Setting the `HandleVisibility` property to `callback` ensures that the figure can be accessed only from within a GUI callback, and cannot be drawn into or deleted from the command line.
- The `Color` property defines the background color of the figure. In this case, it is set to be the same as the default background color of `uicontrol` objects, such as the **Update** push button. The factory default background color of `uicontrol` objects is the system default and can vary from system to system. This statement ensures that the figure's background color matches the background color of the components.

See the Figure Properties reference page for information about figure properties and their default values.

The Axes

The following statement creates the axes.

```
hPlotAxes = axes(...    % Axes for plotting the selected plot
                 'Parent', hMainFigure, ...
                 'Units', 'normalized', ...
                 'HandleVisibility','callback', ...
                 'Position',[0.11 0.13 0.80 0.67]);
```

- The `axes` function creates the axes. Setting the axes `Parent` property to `hMainFigure` makes it a child of the main figure.
- Setting the `Units` property to `normalized` ensures that the axes resizes proportionately when the GUI is resized.
- The `Position` property is a 4-element vector that specifies the location of the axes within the figure and its size: [distance from left, distance from bottom, height, width]. Because the units are normalized, all values are between 0 and 1.

Note If you specify the `Units` property, then the `Position` property, and any other properties that depend on the value of the `Units` property, should follow the `Units` property specification.

See the Axes Properties reference page for information about axes properties and their default values.

The Pop-Up Menu

The following statement creates the pop-up menu.

```
hPlotsPopupmenu = uicontrol(... % List of available types of plot
                      'Parent', hMainFigure, ...
                      'Units','normalized',...
                      'Position',[0.11 0.85 0.45 0.1],...
                      'HandleVisibility','callback', ...
                      'String',mPlotTypes(:,1),...
                      'Style','popupmenu');
```

- The `uicontrol` function creates various user interface controls based on the value of the `Style` property. Here the `Style` property is set to `popupmenu`.
- For a pop-up menu, the `String` property defines the list of items in the menu. Here it is defined as a 5-by-1 cell array of strings derived from the cell array `mPlotTypes`.

See the Uicontrol Properties reference page for information about properties of `uicontrol` objects and their default values.

The Update Push Button

This statement creates the **Update** push button as a uicontrol object.

```
hUpdateButton = uicontrol(... % Button for updating selected plot
                    'Parent', hMainFigure, ...
                    'Units','normalized',...
                    'HandleVisibility','callback', ...
                    'Position',[0.6 0.85 0.3 0.1],...
                    'String','Update',...
                    'Callback', @hUpdateButtonCallback);
```

- The `uicontrol` function creates various user interface controls based on the value of the `Style` property. This statement does not set the `Style` property because its default is `pushbutton`.
- For a push button, the `String` property defines the label on the button. Here it is defined as the string `Update`.
- Setting the `Callback` property to `@hUpdateButtonCallback` defines the name of the callback function that services the push button. That is, clicking the push button triggers the execution of the named callback. This callback function is defined later in the script.

See the Uicontrol Properties reference page for information about properties of `uicontrol` objects and their default values.

The File Menu and Its Menu Items

These statements define the **File** menu and the three items it contains.

```
hFileMenu      =   uimenu(...       % File menu
                       'Parent',hMainFigure,...
                       'HandleVisibility','callback', ...
                       'Label','File');
hOpenMenuitem  =   uimenu(...       % Open menu item
                       'Parent',hFileMenu,...
                       'Label','Open',...
                       'HandleVisibility','callback', ...
                       'Callback', @hOpenMenuitemCallback);
hPrintMenuitem  =  uimenu(...       % Print menu item
                       'Parent',hFileMenu,...
                       'Label','Print',...
                       'HandleVisibility','callback', ...
                       'Callback', @hPrintMenuitemCallback);
hCloseMenuitem  =  uimenu(...       % Close menu item
                       'Parent',hFileMenu,...
                       'Label','Close',...
                       'Separator','on',...
                       'HandleVisibility','callback', ...
                       'Callback', @hCloseMenuitemCallback');
```

- The `uimenu` function creates both the main menu, **File**, and the items it contains. For the main menu and each of its items, set the `Parent` property to the handle of the desired parent to create the menu hierarchy you want.

Here, setting the `Parent` property of the **File** menu to `hMainFigure` makes it the child of the main figure. This statement creates a menu bar in the figure and puts the **File** menu on it.

For each of the menu items, setting its `Parent` property to the handle of the parent menu, `hFileMenu`, causes it to appear on the **File** menu.

- For the main menu and each item on it, the `Label` property defines the strings that appear in the menu.
- Setting the `Separator` property to `on` for the **Close** menu item causes a separator line to be drawn above this item.
- For each of the menu items, the `Callback` property specifies the callback that services that item. In this example, no callback services the **File** menu itself. These callbacks are defined later in the script.

See the Uicontrol Properties reference page for information about properties of `uicontrol` objects and their default values.

The Toolbar and Its Tools

These statements define the toolbar and the two buttons it contains.

```
hToolbar = uitoolbar(...    % Toolbar for Open and Print buttons
                   'Parent',hMainFigure, ...
                   'HandleVisibility','callback');
hOpenPushtool  =  uipushtool(...   % Open toolbar button
                   'Parent',hToolbar,...
                   'TooltipString','Open File',...
                   'CData',iconRead(fullfile(matlabroot,...
                       'toolbox\matlab\icons\opendoc.mat')),...
                   'HandleVisibility','callback', ...
                   'ClickedCallback', @hOpenMenuitemCallback);
hPrintPushtool = uipushtool(...    % Print toolbar button
                    'Parent',hToolbar,...
                    'TooltipString','Print Figure',...
                    'CData',iconRead(fullfile(matlabroot,...
                       'toolbox\matlab\icons\printdoc.mat')),...
                    'HandleVisibility','callback', ...
                    'ClickedCallback', @hPrintMenuitemCallback);
```

- The `uitoolbar` function creates the toolbar on the main figure.
- The `uipushtool` function creates the two push buttons on the toolbar.

- The uipushtool `TooltipString` property assigns a tool tip that displays when the GUI user moves the mouse pointer over the button and leaves it there.
- The `CData` property specifies a truecolor image that displays on the button. For these two buttons, the utility `iconRead` function supplies the image. If you are reading this in the MATLAB Help browser, click here to display this utility M-file in the MATLAB Editor.
- For each of the uipushtools, the `ClickedCallback` property specifies the callback that executes when the GUI user clicks the button. Note that the Open push button and the Print push button use the same callbacks as their counterpart menu items.

See the Uitoolbar Properties reference page for information about properties of `uitoolbar` objects and their default values. See the Uipushtool Properties reference page for information about properties of `uipushtool` objects.

Initializing the GUI

These statements create the plot that appears in the GUI when it first displays, and define the output that is returned to the user if the user provides an output argument when running the GUI.

```
% Update the plot with the initial plot type
localUpdatePlot();

% Define default output and return it if it is requested by users
mOutputArgs{1} = hMainFigure;
if nargout>0
    [varargout{1:nargout}] = mOutputArgs{:};
end
```

- The `localUpdatePlot` function plots the selected plot type in the axes. For a pop-up menu, the uicontrol `Value` property specifies the index of the selected menu item in the `String` property. Since the default value is 1, the initial selection is `'plot(rand(5))'`. The `localUpdatePlot` function is a helper function that is defined later in the script, at the same level as the callbacks.
- The default output is the handle of the main figure.

Defining the Callbacks

This topic defines the callbacks that service the components of the GUI. Because the callback definitions are at a lower level than the component definitions and the data created for the GUI, they have access to all data and component handles.

Although the GUI has six components that are serviced by callbacks, there are only four callback functions. This is because the **Open** menu item and the Open toolbar button share the same callbacks. Similarly, the **Print** menu item and the Print toolbar button share the same callbacks.

- "Update Button Callback" on page 12-18
- "Open Menu Item Callback" on page 12-19
- "Print Menu Item Callback" on page 12-20
- "Close Menu Item Callback" on page 12-20

Note These are the callbacks that were specified in the component definitions, "Creating the GUI and Its Components" on page 12-12.

Update Button Callback

The `hUpdateButtonCallback` function services the **Update** push button. Clicking the Update button triggers the execution of this callback function.

```
function hUpdateButtonCallback(hObject, eventdata)
  % Callback function run when the Update button is pressed
      localUpdatePlot();
  end
```

The `localUpdatePlot` function is a helper function that plots the selected plot type in the axes. It is defined later in the script, "Helper Function: Plotting the Plot Types" on page 12-21.

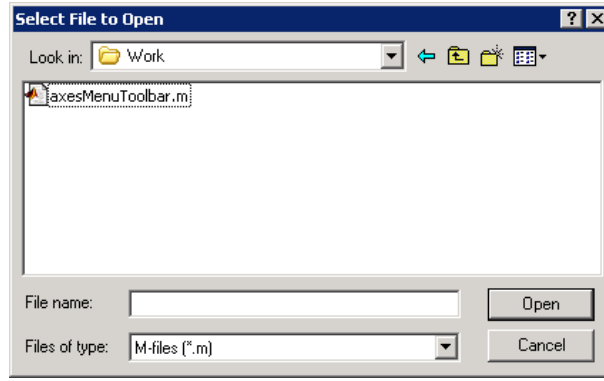
Note MATLAB automatically passes `hUpdateButtonCallback` two arguments, `hObject` and `eventdata`, because the **Update** push button component `Callback` property, `@hUpdateButtonCallback`, is defined as a function handle. `hObject` contains the handle of the component that triggered execution of the callback. `eventdata` is reserved for future use. The function definition line for your callback must account for these two arguments.

Open Menu Item Callback

The `hOpenMenuitemCallback` function services the **Open** menu item and the Open toolbar button . Selecting the menu item or clicking the toolbar button triggers the execution of this callback function.

```
function hOpenMenuitemCallback(hObject, eventdata)
% Callback function run when the Open menu item is selected
    file = uigetfile('*.m');
    if ~isequal(file, 0)
        open(file);
    end
end
```

The `hOpenMenuitemCallback` function first calls the `uigetfile` function to open the standard dialog box for retrieving files. This dialog box lists all M-files. If `uigetfile` returns a filename, the function then calls the `open` function to open it.

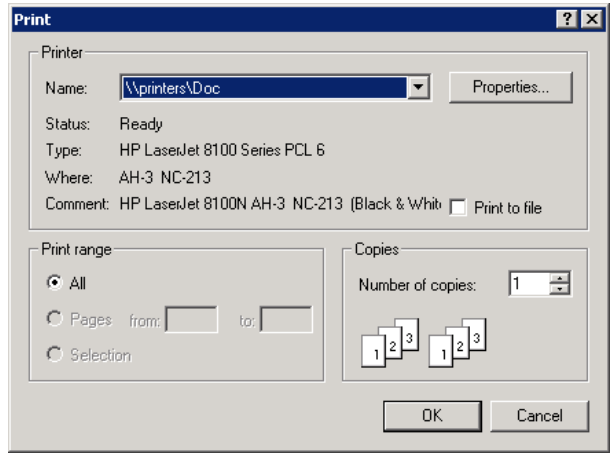


Print Menu Item Callback

The `hPrintMenuitemCallback` function services the **Print** menu item and the Print toolbar button. Selecting the menu item or clicking the toolbar button triggers the execution of this callback function.

```
function hPrintMenuitemCallback(hObject, eventdata)
% Callback function run when the Print menu item is selected
    printdlg(hMainFigure);
end
```

The `hPrintMenuitemCallback` function calls the `printdlg` function. This function opens the standard dialog box for printing the current figure.



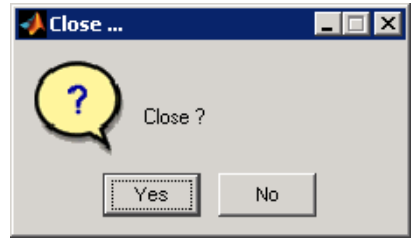
Close Menu Item Callback

The `hCloseMenuitemCallback` function services the **Close** menu item. It executes when the GUI user selects **Close** from the **File** menu.

```
function hCloseMenuitemCallback(hObject, eventdata)
% Callback function run when the Close menu item is selected
    selection = ...
       questdlg(['Close ' get(hMainFigure,'Name') '?'],...
                ['Close ' get(hMainFigure,'Name') '...'],...
                'Yes','No','Yes');
```

```
    if strcmp(selection,'No')
        return;
    end

    delete(hMainFigure);
end
```

The `hCloseMenuitemCallback` function calls the `questdlg` function to create and open the question dialog box shown in the following figure.



If the user clicks the **No** button, the callback returns. If the user clicks the **Yes** button, the callback deletes the GUI.

See “Helper Function: Plotting the Plot Types” on page 12-21 for a description of the `localUpdatePlot` function.

Helper Function: Plotting the Plot Types

The example defines the `localUpdatePlot` function at the same level as the callback functions. Because of this, `localUpdatePlot` has access to the same data and component handles.

```
function localUpdatePlot
% Helper function for plotting the selected plot type
    mPlotTypes{get(hPlotsPopupmenu, 'Value'), 2}(hPlotAxes);
end
```

The `localUpdatePlot` function uses the pop-up menu `Value` property to identify the selected menu item from the first column of the `mPlotTypes` 5-by-2 cell array, then calls the corresponding anonymous function from column two of the cell array to create the plot in the axes.

Color Palette

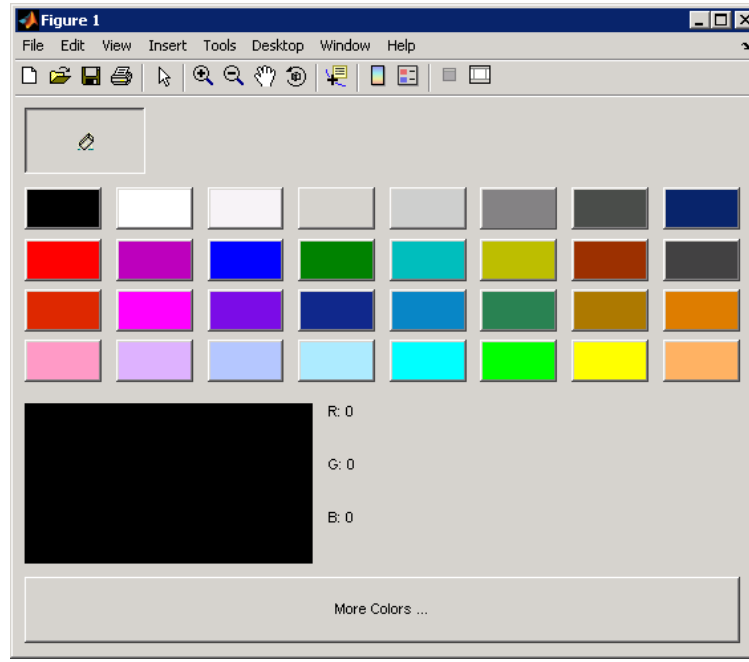
The color palette example includes these topics:

- "The Example" on page 12-22
- "Techniques Used in the Example" on page 12-25
- "View the Completed GUI M-File" on page 12-25
- "Subfunction Summary" on page 12-26
- "M-File Structure" on page 12-27
- "GUI Programming Techniques" on page 12-28

The Example

This example creates a GUI, colorPalette, that enables a user to select a color from a color palette or display the standard color selection dialog box. Another example, "Icon Editor" on page 12-33, embeds the colorPalette, as the child of a panel, in a GUI you can use to design an icon.

The `colorPalette` function populates a GUI figure or panel with a color palette. The figure below shows the palette as the child of a figure.



The Components

The colorPalette includes the following components:

- An array of color cells defined as toggle buttons
- An Eraser toggle button
- A button group that contains the array of color cells and the eraser button. The button group provides exclusive management of these toggle buttons.
- A **More Colors** push button
- A preview of the selected color, below the color cells, defined as a text component
- Text components to specify the red, blue, and green color values

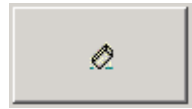
Using the Color Palette

These are the basic steps for using the color palette.

1 Clicking a color cell:

- Displays the selected color in the preview area.
- The red, green, and blue values for the newly selected color are displayed in the R, G, and B fields to the right of the preview area.
- Causes `colorPalette` to return a function handle that the host GUI can use to get the currently selected color.

2 Clicking the Eraser button, causes colorPalette to return a value, `NaN`, that the host GUI can use to remove color from a data point.



3 Clicking the **More Colors** button displays the standard dialog box for setting a color.

Calling the colorPalette Function

You can call the `colorPalette` function with a statement such as

```
mGetColorFcn = colorPalette('Parent',hPaletteContainer)
```

The `colorPalette` function accepts property value pairs as input arguments. Only the custom property `Parent` is supported. This property specifies the handle of the parent figure or panel that contains the color palette. If the call to `colorPalette` does not specify a parent, it uses the current figure, `gcf`. Unrecognized property names or invalid values are ignored.

`colorPalette` returns a function handle that the host GUI can call to get the currently selected color. The host GUI can use the returned function handle at any time before the color palette is destroyed. For more information, see "Sharing Data Between Two GUIs" on page 12-30 for implementation details. "Icon Editor" on page 12-33 is an example of a host GUI that uses the colorPalette.

Techniques Used in the Example

This example illustrates the following techniques:

- Retrieving output from the GUI when it returns.
- Supporting custom input property/value pairs with data validation.
- Sharing data between two GUIs

See "Icon Editor" on page 12-33 for examples of these and other programming techniques.

Note This example uses nested functions. For information about using nested functions, see Nested Functions in the MATLAB documentation.

View the Completed GUI M-File

If you are reading this in the MATLAB Help browser, you can click the following link to display the MATLAB Editor with a complete listing of the code that is discussed in the following sections.

Note The following link executes MATLAB commands and is designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

- Click here to display the main GUI M-file in the MATLAB Editor.

Subfunction Summary

The color palette example includes the callbacks listed in the following table.

Function	Description
`colorCellCallback`	Called by `hPalettePanelSelectionChanged` when any color cell is clicked.
`eraserToolCallback`	Called by `hPalettePanelSelectionChanged` when the Eraser button is clicked.
`hMoreColorButtonCallback`	Executes when the **More Colors** button is clicked. It calls `uisetcolor` to open the standard color-selection dialog box, and calls `localUpdateColor` to update the preview.
`hPalettePanelSelectionChanged`	Executes when the GUI user clicks on a new color. This is the `SectionChangeFcn` callback of the `uibuttongroup` that exclusively manages the tools and color cells that it contains. It calls the appropriate callback to service each of the tools and color cells.

Note Three `eventdata` fields are defined for use with button groups (`uibuttongroup`). These fields enable you to determine the previous and current radio or toggle button selections maintained by the button group. See `SelectionChangeFcn` in the Uibuttongroup Properties for more information.

The example also includes the helper functions listed in the following table.

Function	Description
`layoutComponent`	Dynamically creates the Eraser tool and the color cells in the palette. It calls `localDefineLayout`.
`localUpdateColor`	Updates the preview of the selected color.
`getSelectedColor`	Returns the currently selected color which is then returned to the `colorPalette` caller.
`localDefineLayout`	Calculates the preferred color cell and tool sizes for the GUI. It calls `localDefineColors` and `localDefineTools`
`localDefineTools`	Defines the tools shown in the palette. In this example, the only tool is the Eraser button.
`localDefineColors`	Defines the colors that are shown in the array of color cells.
`processUserInputs`	Determines if the property in a property/value pair is supported. It calls `localValidateInput`.
`localValidateInput`	Validates the value in a property/value pair.

M-File Structure

The colorPalette is programmed using nested functions. Its M-file is organized in the following sequence:

1 Comments displayed in response to the `help` command.

2. Data creation. Because the example uses nested functions, defining this data at the top level makes the data accessible to all functions without having to pass them as arguments.

3. Command line input processing.

4. GUI figure and component creation.

5. GUI initialization.

6. Return output if it is requested.

7. Callback definitions. These callbacks, which service the GUI components, are subfunctions of the `colorPalette` function and so have access to the data and component handles created at the top level, without their having to be passed as arguments.

8. Helper function definitions. These helper functions are subfunctions of the `colorPalette` function and so have access to the data and component handles created at the top level, without their having to be passed as arguments.

Note For information about using nested functions, see Nested Functions in the MATLAB documentation.

GUI Programming Techniques

This topic explains the following GUI programming techniques as they are used in the creation of the colorPalette.

- "Passing Input Arguments to a GUI" on page 12-29
- "Passing Output to a Caller on Returning" on page 12-30
- "Sharing Data Between Two GUIs" on page 12-30

See "Icon Editor" on page 12-33 for additional examples of these and other programming techniques.

Passing Input Arguments to a GUI

Inputs to the GUI are custom property/value pairs. `colorPalette` allows one such property: `Parent`. The names are caseinsensitive. The `colorPalette` syntax is

```
mGetColorFcn = colorPalette('Parent',hPaletteContainer)
```

Definition and Initialization of the Properties. The `colorPalette` function first defines a variable `mInputArgs` as `varargin` to accept the user input arguments.

```
mInputArgs = varargin;  % Command line arguments when invoking
                        % the GUI
```

The `colorPalette` function then defines the valid custom properties in a 3-by-3 cell array.

```
mPropertyDefs = {...    % The supported custom property/value
                        % pairs of this GUI
                'parent',   @localValidateInput, 'mPaletteParent';
```

- The first column contains the property name.
- The second column contains a function handle for the function, `localValidateInput`, that validates the input property values.
- The third column is the local variable that holds the value of the property.

`colorPalette` then initializes the properties with default values.

```
mPaletteParent = [];  % Use input property 'parent' to initialize
```

Processing the Input Arguments. The `processUserInputs` helper function processes the input property/value pairs. `colorPalette` calls `processUserInputs` before it creates the components, to determine the parent of the components.

```
% Process the command line input arguments supplied when
% the GUI is invoked
processUserInputs();
```

1. `processUserInputs` sequences through the inputs, if any, and tries to match each property name to a string in the first column of the `mPropertyDefs` cell array.

2 If it finds a match, `processUserInputs` assigns the value that was input for the property to its variable in the third column of the `mPropertyDefs` cell array.

3 `processUserInputs` then calls the helper function specified in the second column of the `mPropertyDefs` cell array to validate the value that was passed in for the property.

Passing Output to a Caller on Returning

If a host GUI calls the `colorPalette` function with an output argument, it returns a function handle that the host GUI can call to get the currently selected color.

The host GUI calls `colorPalette` only once. The call creates the color palette in the specified parent and then returns the function handle. The host GUI can call the returned function at any time before the color palette is destroyed.

The data definition section of the `colorPalette` M-file creates a cell array to hold the output:

```
mOutputArgs = {};  % Variable for storing output when GUI returns
```

Just before returning, `colorPalette` assigns the function handle, `mgetSelectedColor`, to the cell array `mOutputArgs` and then assigns `mOutputArgs` to `varargout` to return the arguments.

```
mOutputArgs{} = @getSelectedColor;
if nargout>0
    [varargout{1:nargout}] = mOutputArgs{:};
end
```

Sharing Data Between Two GUIs

The iconEditor embeds a GUI, the colorPalette, to enable the user to select colors for the icon cells. The colorPalette returns a function handle the iconEditor. The iconEditor can then call the returned function at any time to get the selected color.

The colorPalette GUI. The `colorPalette` function defines a cell array, `mOutputArgs`, to hold its output arguments.

```
mOutputArgs = {};  % Variable for storing output when GUI returns
```

Just before returning, `colorPalette` assigns `mOutputArgs` the function handle for its `getSelectedColor` helper function and then assigns `mOutputArgs` to `varargout` to return the arguments.

```
% Return user defined output if it is requested
mOutputArgs{1} =@getSelectedColor;
if nargout>0
    [varargout{1:nargout}] = mOutputArgs{:};
end
```

The iconEditor executes the colorPalette's `getSeclectedColor` function whenever it invokes the function that colorPalette returns to it.

```
function color = getSelectedColor
% function returns the currently selected color in this
% colorPlatte
    color = mSelectedColor;
```

The iconEditor GUI. The `iconEditor` function calls `colorPalette` only once and specifies its parent to be a panel in the iconEditor.

```
% Host the ColorPalette in the PaletteContainer and keep the
% function handle for getting its selected color for editing
% icon.
mGetColorFcn = colorPalette('parent', hPaletteContainer);
```

This call creates the colorPalette as a component of the iconEditor and then returns a function handle that `iconEditor` can call to get the currently selected color.

The iconEditor's `localEditColor` helper function calls `mGetColorFcn`, the function returned by `colorPalette`, to execute the colorPalette's `getSelectedColor` function.

```
function localEditColor
% helper function that changes the color of an icon data
% point to that of the currently selected color in
% colorPalette
    if mIsEditingIcon
        pt = get(hIconEditAxes,'currentpoint');
        x = ceil(pt(1,1));
        y = ceil(pt(1,2));
        color = mGetColorFcn();
```

```
            % update color of the selected block
            mIconCData(y, x,:) = color;

            localUpdateIconPlot();
        end
    end
```

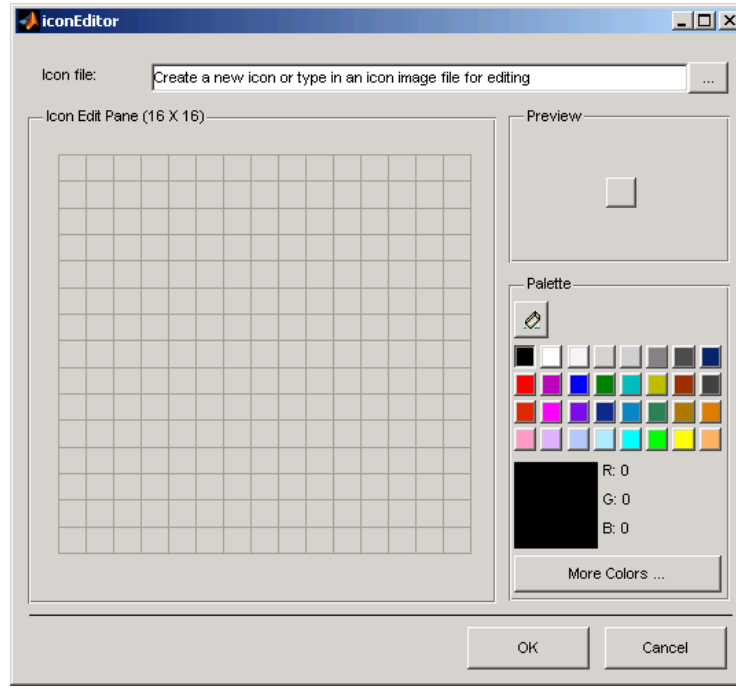
Icon Editor

The icon editor example includes these topics:

- "The Example" on page 12-33
- "Techniques Used in the Example" on page 12-35
- "View the Completed GUI M-Files" on page 12-35
- "Subfunction Summary" on page 12-36
- "M-File Structure" on page 12-37
- "GUI Programming Techniques" on page 12-38

The Example

This example creates a GUI that enables the user to create or edit an icon. The figure below shows the editor.



The Components

The GUI includes the following components:

- A edit text that instructs the user or contains the name of the file to be edited. The edit text is labeled using a static text.
- A push button to the right of the edit text enables the user to select an existing icon file for editing.
- A panel containing an axes. The axes displays a 16-by-16 grid for drawing an icon.
- A panel containing a button that shows a preview of the icon as it is being created.
- A color palette that is created in a separate script and embedded in this GUI. See “Color Palette” on page 12-22.
- A panel, configured as a line, that separates the icon editor from the **OK** and **Cancel** buttons.
- An **OK** push button that causes the GUI to return the icon as an m-by-n-by-3 array and closes the GUI.
- A **Cancel** push button that closes the GUI without returning the icon.

Using the Icon Editor

These are the basic steps to create an icon:

1. Start the icon editor with a command such as

   ```
   myicon = iconEditor('iconwidth',32,'iconheight',56);
   ```

   where the `iconwidth` and `iconheight` properties specify the icon size in pixels.

2. Color the squares in the grid.
   - Click a color cell in the palette. That color is then displayed in the palette preview.
   - Click in specific squares of the grid to transfer the selected color to those squares.
   - Hold down the left mouse button and drag the mouse over the grid to transfer the selected color to the squares that you touch.
   - Change a color by writing over it with another color.

3 Erase the color in some squares.
   - Click the Eraser button on the palette
   - Click in specific squares to erase those squares
   - Click and drag the mouse to erase the squares that you touch.
   - Click a color cell to disable the Eraser.

4 Click **OK** to close the GUI and return, in `myicon`, the icon you created – as a 32-by-65-by-3 array. Click **Cancel** to close the GUI and return an empty array `[]` in `myicon`.

Techniques Used in the Example

This example illustrates the following GUI programming techniques:

- Creating a GUI that does not return a value until the user makes a choice.
- Retrieving output from the GUI when it returns.
- Supporting custom input property/value pairs with data validation.
- Protecting a GUI from being changed from the command line.
- Creating a GUI that runs on multiple platforms
- Sharing data between two GUIs
- Achieving the proper resize behavior

Note This example uses nested functions. For information about using nested functions, see Nested Functions in the MATLAB documentation.

View the Completed GUI M-Files

If you are reading this in the MATLAB Help browser, you can click the following links to display the MATLAB Editor with a complete listing of the code that is discussed in the following sections.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

- Click here to display the main GUI M-file in the MATLAB Editor.
- Click here to display the utility `iconRead` M-file in the MATLAB Editor.

Subfunction Summary

The icon editor example includes the callbacks listed in the following table.

Function	Description
`hMainFigureWindowButtonDownFcn`	Executes when the user clicks a mouse button anywhere in the GUI figure. It calls `localEditColor`.
`hMainFigureWindowButtonUpFcn`	Executes when the user releases the mouse button.
`hMainFigureWindowButtonMotionFcn`	Executes when the user drags the mouse anywhere in the figure with a button pressed. It calls `localEditColor`.
`hIconFileEditCallback`	Executes after the user manually changes the filename of the icon to be edited. It calls `localUpdateIconPlot`.
`hIconFileEditButtondownFcn`	Executes the first time the user clicks the **Icon file** edit box.
`hOKButtonCallback`	Executes when the user clicks the **Ok** push button.

Function	Description
`hCancelButtonCallback`	Executes when the user clicks the **Cancel** push button.
`hIconFileButtonCallback`	Executes when the user clicks the **Icon file** push button . It calls `localUpdateIconPlot`.

The example also includes the helper functions listed in the following table.

Function	Description
`localEditColor`	Changes the color of an icon data point to the currently selected color. Call the function `mGetColorFcn` returned by the `colorPalette` function. It also calls `localUpdateIconPlot`.
`localUpdateIconPlot`	Updates the icon preview. It also updates the axes when an icon is read from a file.
`processUserInputs`	Determines if the property in a property/value pair is supported. It calls `localValidateInput`.
`localValidateInput`	Validates the value in a property/value pair.
`prepareLayout`	Makes changes needed for look and feel and for running on multiple platforms.

M-File Structure

The iconEditor is programmed using nested functions. Its M-file is organized in the following sequence:

1. Comments displayed in response to the `help` command.
2. Data creation. Because the example uses nested functions, defining this data at the top level makes the data accessible to all functions without having to pass them as arguments.

3 GUI figure and component creation.

4 Command line input processing.

5 GUI initialization.

6 Block execution of the program until the GUI user clicks **OK** or **Cancel**.

7 Return output if requested.

8 Callback definitions. These callbacks, which service the GUI components, are subfunctions of the `iconEditor` function and so have access to the data and component handles created at the top level, without their having to be passed as arguments.

9 Helper function definitions. These helper functions are subfunctions of the `iconEditor` function and so have access to the data and component handles created at the top level, without their having to be passed as arguments.

Note For information about using nested functions, see Nested Functions in the MATLAB documentation.

GUI Programming Techniques

This topic explains the following GUI programming techniques as they are used in the creation of the iconEditor.

- "Returning Only After the User Makes a Choice" on page 12-39
- "Passing Input Arguments to a GUI" on page 12-40
- "Retrieving Output on Return from a GUI" on page 12-41
- "Protecting a GUI from Inadvertent Access" on page 12-41
- "Running a GUI on Multiple Platforms" on page 12-42
- "Making a GUI Modal" on page 12-44
- "Sharing Data Between Two GUIs" on page 12-44
- "Achieving Proper Resize Behavior" on page 12-45

Returning Only After the User Makes a Choice

At the end of the initialization code, and just before returning, iconEditor calls `uiwait` with the handle of the main figure to make the GUI blocking.

```
% Make the GUI blocking
uiwait(hMainFigure);

% Return the edited icon CData if it is requested
mOutputArgs{1} =hMainFigure;
mOutputArgs{2} =mIconCData;
if nargout>0
    [varargout{1:nargout}] = mOutputArgs{:};
end
```

Placement of the call to `uiwait` is important. Calling `uiwait` stops the sequential execution of the `iconEdit` M-file after the GUI is initialized and just before the file would return the edited icon data.

When the user clicks the **OK** button, its callback, `hOKButtonCallback`, calls `uiresume` which enables the M-file to resume execution where it stopped and return the edited icon data.

```
function hOKButtonCallback(hObject, eventdata)
% Callback called when the OK button is pressed
    uiresume;
    delete(hMainFigure);
end
```

When the user clicks the **Cancel** button, its callback, `hOCancelButtonCallback`, effectively deletes the icon data then calls `uiresume`. This enables the M-file to resume execution where it stopped but it returns a null matrix.

```
function hCancelButtonCallback(hObject, eventdata)
% Callback called when the Cancel button is pressed
    mIconCData =[];
    uiresume;
    delete(hMainFigure);
end
```

Passing Input Arguments to a GUI

Inputs to the GUI are custom property/value pairs. `iconEdit` allows three such properties: `IconWidth`, `IconHeight`, and `IconFile`. The names are caseinsensitive.

Definition and Initialization of the Properties. The `iconEdit` first defines a variable `mInputArgs` as `varargin` to accept the user input arguments.

```
mInputArgs = varargin;  % Command line arguments when invoking
                        % the GUI
```

The `iconEdit` function then defines the valid custom properties in a 3-by-3 cell array.

```
mPropertyDefs = {...    % Supported custom property/value
                        % pairs of this GUI
                'iconwidth',   @localValidateInput, 'mIconWidth';
               'iconheight',  @localValidateInput, 'mIconHeight';
                'iconfile',    @localValidateInput, 'mIconFile'};
```

- The first column contains the property name.
- The second column contains a function handle for the function, `localValidateInput`, that validates the input property values.
- The third column is the local variable that holds the value of the property.

`iconEdit` then initializes the properties with default values.

```
mIconWidth = 16;   % Use input property 'iconwidth' to initialize
mIconHeight = 16;  % Use input property 'iconheight' to initialize
mIconFile = fullfile(matlabroot,'/toolbox/matlab/icons/');
```

The values of `mIconWidth` and `mIconHeight` are interpreted as pixels. The `fullfile` function builds a full filename from parts.

Processing the Input Arguments. The `processUserInputs` helper function processes the input property/value pairs. `iconEdit` calls `processUserInputs` after the layout is complete and just before it needs the inputs to initialize the GUI.

```
% Process the command line input arguments supplied when
% the GUI is invoked
processUserInputs();
```

1. `processUserInputs` sequences through the inputs, if any, and tries to match each property name to a string in the first column of the `mPropertyDefs` cell array.

2. If it finds a match, `processUserInputs` assigns the value that was input for the property to its variable in the third column of the `mPropertyDefs` cell array.

3. `processUserInputs` then calls the helper function specified in the second column of the `mPropertyDefs` cell array to validate the value that was passed in for the property.

Retrieving Output on Return from a GUI

If you call `iconEditor` with an output argument, it returns a truecolor image as an n-by-m-by-3 array.

The data definition section of the M-file creates a cell array to hold the output:

```
mOutputArgs = {};  % Variable for storing output when GUI returns
```

Following the call to `uiwait`, which stops the sequential execution of the M-file, iconEdit assigns the constructed icon array, `mIconEdit`, to the cell array `mOutputArgs` and then assigns `mOutputArgs` to `varargout` to return the arguments.

```
mOutputArgs{} =mIconCData;
if nargout>0
    [varargout{1:nargout}] = mOutputArgs{:};
end
```

This code is the last that iconEditor executes before returning. It executes only after clicking the **OK** or **Cancel** button triggers execution of `hOKButtonCallback` or `hCancelButtonCallback`, which call `uiresume` to resume execution.

Protecting a GUI from Inadvertent Access

The `prepareLayout` utility function protects the iconEditor from inadvertently being altered from the command line by setting the `HandleVisibility` properties of all the components. The iconEditor calls `prepareLayout` with the handle of the main figure, in the initialization section of the M-file.

```
% Make changes needed for proper look and feel and running on
```

```
% different platforms
prepareLayout(hMainFigure);
```

`prepareLayout` first uses `findall` to retrieve the handles of all objects contained in the figure. The list of retrieved handles includes the `colorPalette`, which is embedded in the `iconEditor`, and its children. The figure's handle is passed to `prepareLayout` as the input argument `topContainer`.

```
allObjects = findall(topContainer);
```

`prepareLayout` then sets the `HandleVisibility` properties of all those objects that have one to `Callback`.

```
% Make GUI objects available to callbacks so that they cannot
% be changed accidentally by other MATLAB commands
set(allObjects(isprop(allObjects,'HandleVisibility')),...
    'HandleVisibility','Callback');
```

Setting `HandleVisibility` to `Callback` causes the GUI handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This ensures that command-line users cannot inadvertently alter the GUI when it is the current figure.

Running a GUI on Multiple Platforms

The `prepareLayout` utility function sets various properties of all the GUI components to enable the GUI to retain the correct look and feel on multiple platforms. The iconEditor calls `prepareLayout` with the handle of the main figure, in the initialization section of the M-file.

```
% Make changes needed for proper look and feel and running on
% different platforms
prepareLayout(hMainFigure);
```

First, `prepareLayout` uses `findall` to retrieve the handles of all objects contained in the figure. The list of retrieved handles also includes the `colorPalette`, which is embedded in the `iconEditor`, and its children. The figure's handle is passed to `findall` as the input argument `topContainer`.

```
function prepareLayout(topContainer)
   ...
   allObjects = findall(topContainer);
```

Background Color. The default component background color is the standard system background color on which the GUI is running. This color varies on different computer systems, e.g., the standard shade of gray on the PC differs from that on UNIX, and may not match the default GUI background color.

The `prepareLayout` function sets the background color of the GUI to be the same as the default component background color. This provides a consistent look within the GUI, as well as with other application GUIs.

It first retrieves the default component background color from the root object. Then sets the GUI background color using the figure's `Color` property.

```
defaultColor = get(0,'defaultuicontrolbackgroundcolor');
if isa(handle(topContainer),'figure')

    ...

    % Make figure color match that of GUI objects
    set(topContainer, 'Color',defaultColor);
end
```

Selecting Units. The `prepareLayout` function decides what units to use based on the GUI's resizability. It uses `strcmpi` to determine the value of the GUI's Resize property. Depending on the outcome, it sets the `Units` properties of all the objects to either `Normalized` or `Characters`.

```
% Make the GUI run properly across multiple platforms by using
% the proper units
if strcmpi(get(topContainer, 'Resize'),'on')
    set(allObjects(isprop(allObjects,'Units')),...
        'Units','Normalized');
else
    set(allObjects(isprop(allObjects,'Units')),...
        'Units','Characters');
end
```

For a resizable figure, normalized units map the lower-left corner of the figure and of each component to (0,0) and the upper-right corner to (1.0,1.0). Because of this, component size is automatically adjusted to its parent's size when the GUI is displayed.

For a nonresizable figure, character units automatically adjusts the size and relative spacing of components as the GUI displays on different computers.

Character units are defined by characters from the default system font. The width of a character unit equals the width of the letter `x` in the system font. The height of a character unit is the distance between the baselines of two lines of text. Note that character units are not square.

Making a GUI Modal

iconEditor is a modal figure. Modal figures remain stacked above all normal figures and the MATLAB command window. This forces the user to respond without being able to interact with other windows. iconEditor makes the main figure modal by setting its `WindowStyle` property to `modal`.

```
hMainFigure = figure(...

 ...

              'WindowStyle', 'modal',...
```

See the Figure Properties in the MATLAB documentation for more information about using the `WindowStyle` property.

Sharing Data Between Two GUIs

The iconEditor embeds a GUI, the colorPalette, to enable the user to select colors for the icon cells. The colorPalette returns the selected color to the iconEditor via a function handle.

The colorPalette GUI. Like the iconEditor, the colorPalette defines a cell array, `mOutputArgs`, to hold its output arguments.

```
mOutputArgs = {};  % Variable for storing output when GUI returns
```

Just before returning, `colorPalette` assigns `mOutputArgs` the function handle for its `getSelectedColor` helper function and then assigns `mOutputArgs` to `varargout` to return the arguments.

```
% Return user defined output if it is requested
mOutputArgs{1} =@getSelectedColor;
if nargout>0
    [varargout{1:nargout}] = mOutputArgs{:};
end
```

The iconEditor executes the colorPalette's `getSeclectedColor` function whenever it invokes the function that colorPalette returns to it.

```
function color = getSelectedColor
% function returns the currently selected color in this
% colorPlatte
    color = mSelectedColor;
```

The iconEditor GUI. The `iconEditor` function calls `colorPalette` only once and specifies its parent to be a panel in the iconEditor.

```
% Host the ColorPalette in the PaletteContainer and keep the
% function handle for getting its selected color for editing
% icon.
mGetColorFcn = colorPalette('parent', hPaletteContainer);
```

This call creates the colorPalette as a component of the iconEditor and then returns a function handle that `iconEditor` can call to get the currently selected color.

The iconEditor's `localEditColor` helper function calls `mGetColorFcn`, the function returned by `colorPalette`, to execute the colorPalette's `getSelectedColor` function.

```
function localEditColor
% helper function that changes the color of an icon data
% point to that of the currently selected color in
% colorPalette
    if mIsEditingIcon
        pt = get(hIconEditAxes,'currentpoint');
        x = ceil(pt(1,1));
        y = ceil(pt(1,2));
        color = mGetColorFcn();

        % update color of the selected block
        mIconCData(y, x,:) = color;

        localUpdateIconPlot();
    end
end
```

Achieving Proper Resize Behavior

The `prepareLayout` utility function sets the `Units` properties of all the GUI components to enable the GUI to resize correctly on multiple platforms. The

iconEditor calls `prepareLayout` with the handle of the main figure, in the initialization section of the M-file.

```
prepareLayout(hMainFigure);
```

First, `prepareLayout` uses `findall` to retrieve the handles of all objects contained in the figure. The list of retrieved handles includes the `colorPalette`, which is embedded in the `iconEditor`, and its children. The figure's handle is passed to `findall` as the input argument `topContainer`.

```
function prepareLayout(topContainer)
   ...
   allObjects = findall(topContainer);
```

Then, `prepareLayout` uses `strcmpi` to determine if the GUI is resizable. Depending on the outcome, it sets the `Units` properties of all the objects to either `Normalized` or `Characters`.

```
if strcmpi(get(topContainer, 'Resize'),'on')
    set(allObjects(isprop(allObjects,'Units')),...
        'Units','Normalized');
else
    set(allObjects(isprop(allObjects,'Units')),...
        'Units','Characters');
end
```

Note The iconEditor is resizable because it accepts the default value, `on`, of the figure `Resize` property.

Resizable Figure. Normalized units map the lower-left corner of the figure and of each component to (0,0) and the upper-right corner to (1.0,1.0). Because of this, when the GUI is resized, component size is automatically changed relative its parent's size.

Nonresizable Figure. Character units automatically adjusts the size and relative spacing of components as the GUI displays on different computers.

Character units are defined by characters from the default system font. The width of a character unit equals the width of the letter `x` in the system font. The

height of a character unit is the distance between the baselines of two lines of text. Note that character units are not square.

Index

A

ActiveX controls
  adding to layout 6-22
  programming 8-31
aligning components
  in GUIDE 6-39
Alignment Tool
  GUIDE 6-39
application data
  `appdata` functions 9-4
application-defined data
  application data 9-4
  GUI data 9-2
  in GUIDE GUIs 9-1
  `UserData` property 9-6
axes
  multiple in GUI 10-2
axes, plotting when hidden 10-28

B

background color
  system standard for GUIs 6-64
backward compatibility
  GUIs to Version 6 5-4
button groups 6-18
  adding components 6-20

C

callback
  arguments 8-9
callback templates (GUIDE)
  add comments 5-8
callbacks
  sharing data 9-7
check boxes 8-21
color of GUI background 5-12
command-line accessibility of GUIs 5-10
compatibility across platforms
  GUI design 6-63
component identifier
  assigning in GUIDE 6-24
component palette
  show names 5-7
component text
  adding in GUIDE 6-25
components for GUIs
  GUIDE 6-14
components in GUIDE
  aligning 6-39
  copying 6-32
  cutting and clearing 6-32
  front-to-back positioning 6-33
  moving 6-34
  pasting and duplicating 6-32
  resizing 6-37
  selecting 6-31
  tab order 6-45
confirmation
  exporting a GUI 5-2
  GUI activation 5-2
context menus
  associating with an object 6-59
  creating in GUIDE 6-48
  creating with GUIDE 6-56
  menu items 6-57
  parent menu 6-56

cross-platform compatibility
GUI background color 6-64
GUI design 6-63
GUI fonts 6-63
GUI units 6-66

D

data
sharing among GUI callbacks 9-7
default system font
in GUIs 6-63

E

edit text 8-22
exporting a GUI
confirmation 5-2

F

FIG-file
generate in GUIDE 5-13
generated by GUIDE 5-11
files
GUIDE GUI 7-2
fixed-width font
in GUIs 6-64
fonts
using specific in GUIs 6-64
function prototypes
GUIDE option 5-11

G

GUI
adding components with GUIDE 6-14
application-defined data (GUIDE) 9-1
command-line arguments 8-15
compatibility with Version 6 5-4
designing 6-2
GUIDE options 5-9
help button 10-29
laying out in GUIDE 6-1
naming in GUIDE 7-2
opening function 8-15
renaming in GUIDE 7-4
resize function 10-44
resizing 5-9
running 7-10
saving in GUIDE 7-5
standard system background color 6-64
using default system font 6-63
with multiple axes 10-2
GUI component text
adding in GUIDE 6-25
GUI components
GUIDE 6-14
how to add in GUIDE 6-18
moving in GUIDE 6-34
tab order in GUIDE 6-45
GUI data
application-defined data 9-2
GUI export
confirmation 5-2
GUI files
in GUIDE 7-2
GUI layout in GUIDE
copying components 6-32
cutting and clearing components 6-32
moving components 6-34
pasting and duplicating components 6-32
selecting components 6-31

GUI object hierarchy
  viewing in GUIDE 6-62
GUI options (GUIDE)
  function prototypes 5-11
  singleton 5-11
  system color background 5-11
GUI size
  setting with GUIDE 6-12
GUI template
  selecting in GUIDE 6-5
GUI units
  cross-platform compatible 6-66
GUIDE
  adding components to GUI 6-14
  application examples 10-1
  application-defined data 9-1
  command-line accessibility of GUIs 5-10
  creating menus 6-48
  generate FIG-file only 5-13
  generated M-file 5-11
  grids and rulers 6-42
  GUI background color 5-12
  GUI files 7-2
  how to add components 6-18
  Object Browser 6-62
  preferences 5-2
  renaming files 7-4
  resizing GUIs 5-9
  saving a GUI 7-5
  selecting template 6-5
  starting 6-4
  tool summary 4-3
  what is 4-2
GUIDE callback templates
  add comments 5-8

H

`handles` structure
  adding fields 9-3
help button for GUIs 10-29
hidden figure, accessing 10-28

I

identifier
  assigning to GUI component 6-24

L

labels
  adding to GUI components in GUIDE 6-25
Layout Editor
  show component names 5-7
Layout Editor window
  show file extension 5-8
  show file path 5-8
list boxes 8-23
  example 10-9

M

menu item
  check 8-38
menus
  callbacks 8-37
  context menus in GUIDE 6-56
  creating in GUIDE 6-48
  drop-down menus 6-49
  menu bar menus 6-49
  menu items 6-51, 6-57
  parent of context menu 6-56
  pop-up 8-24
  specifying properties 6-50

M-file
  generated by GUIDE 5-11
moving components
  in GUIDE 6-34

N

naming a GUI
  in GUIDE 7-2

O

Object Browser (GUIDE) 6-62
opening .fig files 10-14
options
  GUIDE GUIs 5-9

P

panels 6-17
  adding components 6-20
pop-up menus 8-24
preferences
  GUIDE 5-2

R

radio buttons 8-21
renaming GUIDE GUIs 7-4
resize function for GUI 10-44
resizing components
  in GUIDE 6-37
resizing GUIs 5-9
running a GUI 7-10

S

saving GUI
  in GUIDE 7-5
shortcut menus
  creating in GUIDE 6-56
single instance 5-12
singleton GUI
  GUIDE option 5-11
size of GUI
  setting with GUIDE 6-12
sliders 6-17
system color background
  GUIDE option 5-11

T

tab order
  components in GUIDE 6-45
Tab Order Editor 6-45
`Tag` property
  assigning in GUIDE 6-24
template for GUI
  selecting in GUIDE 6-5
toggle buttons 8-20
toolbar
  show in GUIDE Layout Editor 5-6
toolbar menus
  creating with GUIDE 6-49
toolbars
  creating 6-61

U

units for GUIs
  cross-platform compatible 6-66
`UserData` property
  application-defined data 9-6